(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
1 November 2001 (01.11.2001)

PCT

(10) International Publication Number
WO 01/82057 A2

(51) International Patent Classification[7]: G06F 9/00

(21) International Application Number: PCT/US01/12609

(22) International Filing Date: 18 April 2001 (18.04.2001)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

09/551,311	18 April 2000 (18.04.2000)	US
60/261,633	12 January 2001 (12.01.2001)	US

(71) Applicant: **SUN MICROSYSTEMS, INC.** [US/US]; 901 San Antonio Road, Palo Alto, CA 94303 (US).

(72) Inventors: **MOIR, Mark, S.**; 108 Liberty Road, #2, Somerville, MA 02144 (US). **DETLEFS, David, L.**; 94 Depot Street, Westford, MA 01886 (US). **STEELE, Guy, L., Jr.**; 9 Lantern Lane, Lexington, MA 02421 (US). **GARTHWAITE, Alexander, T.**; 2 Burton Avenue, Beverly, MA 01915 (US). **MARTIN, Paul, A.**; 70 Ronald Road, Arlington, MA 02474 (US). **SHAVIT, Nir, N.**; Scrieber Building, Room 220, Tel-Aviv University, Ramat Aviv, 69978 Tel Aviv (IL).

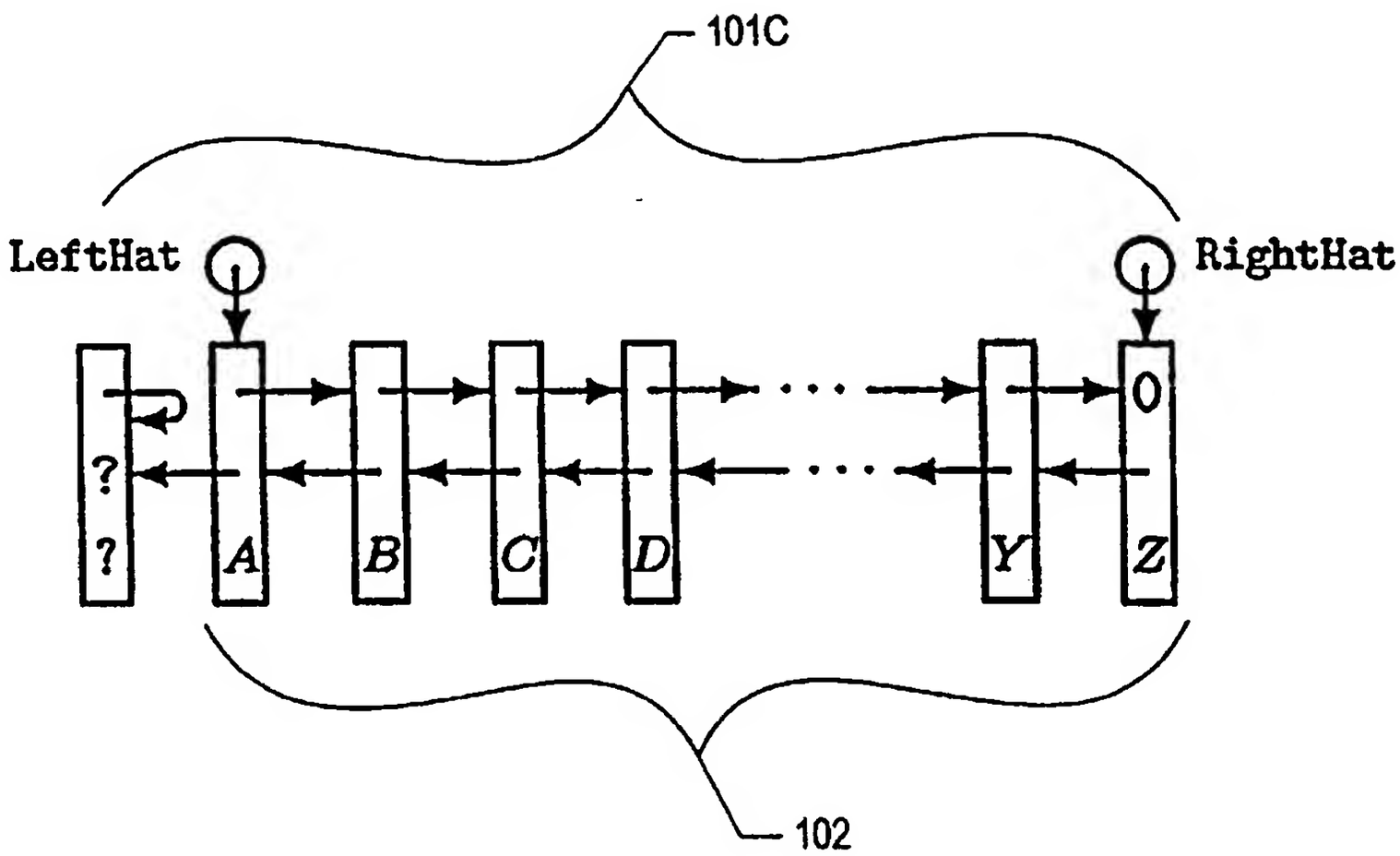
(74) Agents: **O'BRIEN, David, W.** et al.; Zagorin, O'Brien & Graham, LLP, Suite 870, 401 West 15th Street, Austin, TX 78701 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: LOCK-FREE IMPLEMENTATION OF CONCURRENT SHARED OBJECT WITH DYNAMIC NODE ALLOCATION AND DISTINGUISHING POINTER VALUE



(57) Abstract: A novel linked-list-based concurrent shared object implementation has been developed that provides non-blocking and linearizable access to the concurrent shared object. In an application of the underlying techniques to a deque, non-blocking completion of access operations is achieved without restricting concurrency in accessing the deque's two ends. In various realizations in accordance with the present invention, the set of values that may be pushed onto a shared object is not constrained by use of distinguishing values. In addition, an explicit reclamation embodiment facilitates use in environments or applications where automatic reclamation of storage is unavailable or impractical.

WO 01/82057 A2

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

LOCK-FREE IMPLEMENTATION OF CONCURRENT SHARED OBJECT WITH DYNAMIC NODE ALLOCATION AND DISTINGUISHING POINTER VALUE

Technical Field

The present invention relates generally to coordination amongst execution sequences in a multiprocessor computer, and more particularly, to structures and techniques for facilitating non-blocking access to concurrent shared objects.

Background Art

An important abstract data structure in computer science is the "double-ended queue" (abbreviated "deque" and pronounced "deck"), which is a linear sequence of items, usually initially empty, that supports the four operations of inserting an item at the left-hand end ("left push"), removing an item from the left-hand end ("left pop"), inserting an item at the right-hand end ("right push"), and removing an item from the right-hand end ("right pop").

Sometimes an implementation of such a data structure is shared among multiple concurrent processes, thereby allowing communication among the processes. It is desirable that the data structure implementation behave in a linearizable fashion; that is, as if the operations that are requested by various processes are performed atomically in some sequential order.

One way to achieve this property is with a mutual exclusion lock (sometimes called a semaphore). For example, when any process issues a request to perform one of the four deque operations, the first action is to acquire the lock, which has the property that only one process may own it at a time. Once the lock is acquired, the operation is performed on the sequential list; only after the operation has been completed is the lock released. This clearly enforces the property of linearizability.

However, it is generally desirable for operations on the left-hand end of the deque to interfere as little as possible with operations on the right-hand end of the deque. Using a mutual exclusion lock as described above, it is impossible for a request for an operation on the right-hand end of the deque to make any progress while the deque is locked for the purposes of performing an operation on the left-hand end. Ideally, operations on one end of the deque would never impede operations on the other end of the deque unless the deque were nearly empty (containing two items or fewer) or, in some implementations, nearly full.

In some computational systems, processes may proceed at very different rates of execution; in particular, some processes may be suspended indefinitely. In such circumstances, it is highly desirable for the implementation of a deque to be "non-blocking" (also called "lock-free"); that is, if a set of processes are using a deque and an arbitrary subset of those processes are suspended indefinitely, it is always still possible for at least one of the remaining processes to make progress in performing operations on the deque.

Certain computer systems provide primitive instructions or operations that perform compound operations on memory in a linearizable form (as if atomically). The VAX computer, for example, provided instructions to directly support the four deque operations. Most computers or processor architectures provide

simpler operations, such as "test-and-set"; (IBM 360), "fetch-and-add" (NYU Ultracomputer), or "compare-and-swap" (SPARC). SPARC® architecture based processors are available from Sun Microsystems, Inc., Mountain View, California. SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems.

The "compare-and-swap" operation (CAS) typically accepts three values or quantities: a memory address A, a comparison value C, and a new value N. The operation fetches and examines the contents V of memory at address A. If those contents V are equal to C, then N is stored into the memory location at address A, replacing V. Whether or not V matches C, V is returned or saved in a register for later inspection. All this is implemented in a linearizable, if not atomic, fashion. Such an operation may be notated as "CAS(A, C, N)".

Non-blocking algorithms can deliver significant performance benefits to parallel systems. However, there is a growing realization that existing synchronization operations on single memory locations, such as compare-and-swap (CAS), are not expressive enough to support design of efficient non-blocking algorithms. As a result, stronger synchronization operations are often desired. One candidate among such operations is a double-word ("extended") compare-and-swap (implemented as a CASX instruction in some versions of the SPARC architecture), which is simply a CAS that uses operands of two words in length. It thus operates on two memory addresses, but they are constrained to be adjacent to one another. A more powerful and convenient operation is "double compare-and-swap" (DCAS), which accepts six values: memory addresses A1 and A2, comparison values C1 and C2, and new values N1 and N2. The operation fetches and examines the contents V1 of memory at address A1 and the contents V2 of memory at address A2. If V1 equals C1 and V2 equals C2, then N1 is stored into the memory location at address A1, replacing V1, and N2 is stored into the memory location at address A2, replacing V2. Whether or not V1 matches C1 and whether or not V2 matches C2, V1 and V2 are returned or saved in a registers for later inspection. All this is implemented in a linearizable, if not atomic, fashion. Such an operation may be notated as "DCAS(A1, A2, C1, C2, N1, N2)".

Massalin and Pu disclose a collection of DCAS-based concurrent algorithms. See e.g., H. Massalin and C. Pu, A Lock-Free Multiprocessor OS Kernel, Technical Report TR CUCS–005–9, Columbia University, New York, NY, 1991, pages 1-19. In particular, Massalin and Pu disclose a lock-free operating system kernel based on the DCAS operation offered by the Motorola 68040 processor, implementing structures such as stacks, FIFO-queues, and linked lists. Unfortunately, the disclosed algorithms are centralized in nature. In particular, the DCAS is used to control a memory location common to all operations and therefore limits overall concurrency.

Greenwald discloses a collection of DCAS-based concurrent data structures that improve on those of Massalin and Pu. See e.g., M. Greenwald. Non-Blocking Synchronization and System Design, Ph.D. thesis, Stanford University Technical Report STAN-CS-TR-99-1624, Palo Alto, CA, 8 1999, 241 pages. In particular, Greenwald discloses implementations of the DCAS operation in software and hardware and discloses two DCAS-based concurrent double-ended queue (deque) algorithms implemented using an array. Unfortunately, Greenwald's algorithms use DCAS in a restrictive way. The first, described in Greenwald,

Non-Blocking Synchronization and System Design, at pages 196-197, uses a two-word DCAS as if it were a three-word operation, storing two deque end pointers in the same memory word, and performing the DCAS operation on the two-pointer word and a second word containing a value. Apart from the fact that Greenwald's algorithm limits applicability by cutting the index range to half a memory word, it also prevents concurrent access to the two ends of the deque. Greenwald's second algorithm, described in Greenwald, Non-Blocking Synchronization and System Design, at pages 217-220, assumes an array of unbounded size, and does not deal with classical array-based issues such as detection of when the deque is empty or full.

Arora et al. disclose a CAS-based deque with applications in job-stealing algorithms. See e.g., N. S. Arora, Blumofe, and C. G. Plaxton, Thread Scheduling For Multiprogrammed Multiprocessors, in Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures, 1998. Unfortunately, the disclosed non-blocking implementation restricts one end of the deque to access by only a single processor and restricts the other end to only pop operations.

Accordingly, improved techniques are desired that provide linearizable and non-blocking (or lock-free) behavior for implementations of concurrent shared objects such as a deque, and which do not suffer from the above-described drawbacks of prior approaches.

DISCLOSURE OF THE INVENTION

A set of structures and techniques are described herein whereby an exemplary concurrent shared object, namely a double-ended queue (deque), is implemented. Although non-blocking, linearizable deque implementations exemplify several advantages of realizations in accordance with the present invention, the present invention is not limited thereto. Indeed, based on the description herein and the claims that follow, persons of ordinary skill in the art will appreciate a variety of concurrent shared object implementations. For example, although the described deque implementations exemplify support for concurrent push and pop operations at both ends thereof, other concurrent shared objects implementations in which concurrency requirements are less severe, such as LIFO or stack structures and FIFO or queue structures, may also be implemented using the techniques described herein. Accordingly, subsets of the functional sequences and techniques described herein for exemplary deque realizations may be employed to support any of these simpler structures.

Furthermore, although various non-blocking, linearizable deque implementations described herein employ a particular synchronization primitive, namely a double compare and swap (DCAS) operation, the present invention is not limited to DCAS-based realizations. Indeed, a variety of synchronization primitives may be employed that allow linearizable, if not atomic, update of at least a pair of storage locations. In general, N-way Compare and Swap (NCAS) operations ($N \geq 2$) or transactional memory may be employed.

Choice of an appropriate synchronization primitive is typically affected by the set of alternatives available in a given computational system. While direct hardware- or architectural-support for a particular primitive is preferred, software emulations that build upon an available set of primitives may also be suitable

for a given implementation. Accordingly, any synchronization primitive that allows access operations to be implemented with substantially equivalent semantics to those described herein is suitable.

Accordingly, a novel linked-list-based concurrent shared object implementation has been developed that provides non-blocking and linearizable access to the concurrent shared object. In an application of the underlying techniques to a deque, non-blocking completion of access operations is achieved without restricting concurrency in accessing the deque's two ends. In various realizations in accordance with the present invention, the set of values that may be pushed onto a shared object is not constrained by use of distinguishing values. In addition, an explicit reclamation embodiment facilitates use in environments or applications where automatic reclamation of storage is unavailable or impractical.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention may be better understood, and its numerous objects, features, and advantages made apparent to those skilled in the art by referencing the accompanying drawings.

FIGS. 1A, 1B, 1C and **1D** depict several states of a linked-list structure encoding a double-ended queue (deque) in accordance with an exemplary embodiment of the present invention. In particular, **FIG. 1A** depicts an illustrative state after push operations at both ends of the deque. **FIG. 1B** depicts an illustrative state after pop operations at both ends of the deque. **FIG. 1C** depicts an illustrative state after a pop operation at a left end of the deque and a push operation at a right end of the deque. Finally, **FIG. 1D** depicts an illustrative state after a push operation at a left end of the deque and a pop operation at a right end of the deque.

FIGS. 2A, 2B, 2C and **2D** depict several states of a linked-list structure encoding an empty double-ended queue (deque) in accordance with an exemplary embodiment of the present invention. In particular, **FIG. 2A** depicts an empty state. **FIG. 2B** depicts a logically empty state that may result from concurrent execution of pop operations at opposing ends of a deque implemented in accordance with an exemplary embodiment of the present invention. Finally, **FIGS. 2C** and **2D** depict logically empty states that may arise in some embodiments in accordance with the present invention. Other empty deque state encodings are described elsewhere herein.

FIGS. 3A, 3B, 3C and **3D** depict several states of a linked-list structure encoding a single-element double-ended queue (deque) in accordance with an exemplary embodiment of the present invention.

FIGS. 4A, 4B, 4C and **4D** depict several states of a linked-list structure encoding a two-element double-ended queue (deque) in accordance with an exemplary embodiment of the present invention.

FIG. 5 depicts a shared memory multiprocessor configuration that serves as a useful illustrative environment for describing operation of some shared object implementations in accordance with the present invention.

The use of the same reference symbols in different drawings indicates similar or identical items.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

The description that follows presents a set of techniques, objects, functional sequences and data structures associated with concurrent shared object implementations employing linearizable synchronization operations in accordance with an exemplary embodiment of the present invention. An exemplary non-blocking, linearizable concurrent double-ended queue (deque) implementation that employs double compare-and-swap (DCAS) operations is illustrative. A deque is a good exemplary concurrent shared object implementation in that it involves all the intricacies of LIFO-stacks and FIFO-queues, with the added complexity of handling operations originating at both of the deque's ends. Accordingly, techniques, objects, functional sequences and data structures presented in the context of a concurrent deque implementation will be understood by persons of ordinary skill in the art to describe a superset of support and functionality suitable for less challenging concurrent shared object implementations, such as LIFO-stacks, FIFO-queues or concurrent shared objects (including deques) with simplified access semantics.

In view of the above, and without limitation, the description that follows focuses on an exemplary linearizable, non-blocking concurrent deque implementation that behaves as if access operations on the deque are executed in a mutually exclusive manner, despite the absence of a mutual exclusion mechanism. Advantageously, and unlike prior approaches, deque implementations in accordance with some embodiments of the present invention allow concurrent operations on the two ends of the deque to proceed independently. Since synchronization operations are relatively slow and/or impose overhead, it is generally desirable to minimize their use. Accordingly, one advantage of some implementations in accordance with the present invention is that in typical execution paths, access operations require only a single synchronization operation.

Computational Model

One realization of the present invention is as a deque implementation employing the DCAS operation on a shared memory multiprocessor computer. This realization, as well as others, will be understood in the context of the following computation model, which specifies the concurrent semantics of the deque data structure.

In general, a concurrent system consists of a collection of n processors. Processors communicate through shared data structures called objects. Each object has an associated set of primitive operations that provide the mechanism for manipulating that object. Each processor P can be viewed in an abstract sense as a sequential thread of control that applies a sequence of operations to objects by issuing an invocation and receiving the associated response. A history is a sequence of invocations and responses of some system execution. Each history induces a "real-time" order of operations where an operation A precedes another operation B, if A's response occurs before B's invocation. Two operations are concurrent if they are unrelated by the real-time order. A sequential history is a history in which each invocation is followed immediately by its corresponding response. The sequential specification of an object is the set of legal sequential histories associated with it. The basic correctness requirement for a concurrent implementation is linearizability. Every

concurrent history is "equivalent" to some legal sequential history which is consistent with the real-time order induced by the concurrent history. In a linearizable implementation, an operation appears to take effect atomically at some point between its invocation and response. In the model described herein, the collection of shared memory locations of a multiprocessor computer's memory (including location L) is a linearizable implementation of an object that provides each processor P_i with the following set of sequentially specified machine operations:

$Read_i$ (L) reads location L and returns its value.

$Write_i$ (L,v) writes the value v to location L.

$DCAS_i$ (L1, L2, o1, o2, n1, n2) is a double compare-and-swap operation with the semantics described below.

Implementations described herein are *non-blocking* (also called *lock-free*). Let us use the term *higher-level operations* in referring to operations of the data type being implemented, and *lower-level operations* in referring to the (machine) operations in terms of which it is implemented. A non-blocking implementation is one in which, even though individual higher-level operations may be delayed, the system as a whole continuously makes progress. More formally, a non-blocking implementation is one in which any infinite history containing a higher-level operation that has an invocation but no response must also contain infinitely many responses. In other words, if some processor performing a higher-level operation continuously takes steps and does not complete, it must be because some operations invoked by other processors are continuously completing their responses. This definition guarantees that the system as a whole makes progress and that individual processors cannot be blocked, only delayed by other processors continuously taking steps. Using locks would violate the above condition, hence the alternate name: lock-free.

Double Compare-and-Swap Operation

Double compare-and-swap (DCAS) operations are well known in the art and have been implemented in hardware, such as in the Motorola 68040 processor, as well as through software emulation. Accordingly, a variety of suitable implementations exist and the descriptive code that follows is meant to facilitate later description of concurrent shared object implementations in accordance with the present invention and not to limit the set of suitable DCAS implementations. For example, order of operations is merely illustrative and any implementation with substantially equivalent semantics is also suitable. Similarly, some formulations (such as described above) may return previous values while others may return success/failure indications. The illustrative formulation that follows is of the latter type. In general, any of a variety of formulations is suitable.

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1==old1) && (*addr2==old2)) {
      *addr1 = new1;
      *addr2 = new2;
      return true;
```

```
        } else {
          return false;
        }
      }
    }
```

The above sequence of operations implementing the DCAS operation are executed atomically using support suitable to the particular realization. For example, in various realizations, through hardware support (e.g., as implemented by the Motorola 68040 microprocessor or as described in M. Herlihy and J. Moss, Transactional memory: Architectural Support For Lock-Free Data Structures, Technical Report CRL 92/07, Digital Equipment Corporation, Cambridge Research Lab, 1992, 12 pages), through non-blocking software emulation (such as described in G. Barnes, A Method For Implementing Lock-Free Shared Data Structures, in Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures, pages 261–270, June 1993 or in N. Shavit and D. Touitou, Software transactional memory, Distributed Computing, 10(2):99–116, February 1997), or via a blocking software emulation.

Although the above-referenced implementation is presently preferred, other DCAS implementations that substantially preserve the semantics of the descriptive code (above) are also suitable. Furthermore, although much of the description herein is focused on double compare-and-swap (DCAS) operations, it will be understood that other synchronization operations such as N-location compare-and-swap operations ($N \geq 2$) or transactional memory may be more generally employed.

A Double-ended Queue (Deque)

A deque object S is a concurrent shared object, that in an exemplary realization is created by an operation of a constructor operation, e.g., `make_deque()`, and which allows each processor P_i, $0 \leq i \leq n - 1$, of a concurrent system to perform the following types of operations on S: `push_right`$_i$`(v)`, `push_left`$_i$`(v)`, `pop_right`$_i$`()`, and `pop_left`$_i$`()`. Each push operation has an input, v, where v is selected from a range of values. Each pop operation returns an output from the range of values. Push operations on a full deque object and pop operations on an empty deque object return appropriate indications. In the case of a dynamically sized deque, "full" refers to the case where the deque is observed to have no available nodes to accommodate a push and the system storage allocator reports that no more storage is available to the process.

A concurrent implementation of a deque object is one that is linearizable to a standard sequential deque. This sequential deque can be specified using a state-machine representation that captures all of its allowable sequential histories. These sequential histories include all sequences of push and pop operations induced by the state machine representation, but do not include the actual states of the machine. In the following description, we abuse notation slightly for the sake of clarity.

The state of a deque is a sequence of items $S = \langle v_0, \ldots, v_k \rangle$ from the range of values, having cardinality $0 \leq |S| \leq$ `max_length_S`. The deque is initially in the empty state (following invocation of `make_deque()`), that is, has cardinality 0, and is said to have reached a full state if its cardinality is

`max_length_S`. In general, for deque implementations described herein, cardinality is unbounded except by limitations (if any) of an underlying storage allocator.

The four possible push and pop operations, executed sequentially, induce the following state transitions of the sequence $S = \langle v_0,\ldots,v_k\rangle$, with appropriate returned values:

`push_right` (v_{new})	if S is not full, sets S to be the sequence $S = \langle v_0,\ldots,v_k,v_{new}\rangle$
`push_left` (v_{new})	if S is not full, sets S to be the sequence $S = \langle v_{new},v_0,\ldots,v_k\rangle$
`pop_right` ()	if S is not empty, sets S to be the sequence $S = \langle v_0,\ldots,v_{k-1}\rangle$ and returns the item, v_k.
`pop_left` ()	if S is not empty, sets S to be the sequence $S = \langle v_1,\ldots,v_k\rangle$ and returns the item v_0.

For example, starting with an empty deque state, $S = \langle\rangle$, the following sequence of operations and corresponding transitions can occur. A `push_right` (1) changes the deque state to $S = \langle 1\rangle$. A `push_left` (2) subsequently changes the deque state to $S = \langle 2,1\rangle$. A subsequent `push_right` (3) changes the deque state to $S = \langle 2,1,3\rangle$. Finally, a subsequent `pop_right` () changes the deque state to $S = \langle 2,1\rangle$ and returns the value, 3. In some implementations, return values may be employed to indicate success or failure.

Storage Reclamation

Many programming languages and execution environments have traditionally placed responsibility for dynamic allocation and deallocation of memory on the programmer. For example, in the C programming language, memory is allocated from the heap by the `malloc` procedure (or its variants). Given a pointer variable, `p`, execution of machine instructions corresponding to the statement `p=malloc (sizeof (SomeStruct))` causes pointer variable `p` to point to newly allocated storage for a memory object of size necessary for representing a `SomeStruct` data structure. After use, the memory object identified by pointer variable `p` can be deallocated, or freed, by calling `free (p)`. Other languages provide analogous facilities for explicit allocation and deallocation of memory.

Unfortunately, dynamically allocated storage becomes unreachable when no chain of references (or pointers) can be traced from a "root set" of references (or pointers) to the storage. Memory objects that are no longer reachable, but have not been freed, are called garbage. Similarly, storage associated with a memory object can be deallocated while still referenced. In this case, a dangling reference has been created. In general, dynamic memory can be hard to manage correctly. Because of this difficulty, garbage collection, i.e., automatic reclamation of dynamically allocated storage, can be an attractive model of memory management. Garbage collection is particularly attractive for languages such as the JAVA™ language (JAVA and all Java-based marks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries), Prolog, Lisp, Smalltalk, Scheme, Eiffel, Dylan, ML, Haskell, Miranda, Oberon, etc. See generally, Jones & Lins, Garbage Collection: Algorithms for Automatic Dynamic Memory Management, pp.

1-41, Wiley (1996) for a discussion of garbage collection and of various algorithms and implementations for performing garbage collection.

In general, the availability of particular memory management facilities are language, implementation and execution environment dependent. Accordingly, for some realizations in accordance with the present invention, it is acceptable to assume that storage is managed by a "garbage collector" that returns (to a "free pool") that storage for which it can be proven that no process will, in the future, access data structures contained therein. Such a storage management scheme allows operations on a concurrent shared object, such as a deque, to simply eliminate references or pointers to a removed data structure and rely upon operation of the garbage collector for automatic reclamation of the associated storage.

However, for some realizations, a garbage collection facility may be unavailable or impractical. For example, one realization in which automatic reclamation may be unavailable or impractical is a concurrent shared object implementation (e.g., a deque) employed in the implementation of a garbage collector itself. Accordingly, in some realizations in accordance with the present invention, storage is explicitly reclaimed or "freed" when no longer used. For example, in some realizations, removal operations include explicit reclamation of the removed storage.

Linked-List-based Deque Implementation

One embodiment in accordance with the present invention includes a linked-list-based implementation of a lock-free double-ended queue (deque). The implementation includes both structures (e.g., embodied as data structures in memory and/or other storage) and techniques (e.g., embodied as operations, functional sequences, instructions, etc.) that facilitate concurrent, non-blocking shared access. The exemplary implementation employs double compare and swap (DCAS) operations to provide linearizable behavior. However, as described elsewhere herein, other synchronization primitives may be employed in other realizations. In general, the exemplary implementation exhibits a number of features that tend to improve its performance:

a) Access operations (e.g., push and pop operations) at opposing left and right ends of the deque do not interfere with each other except when the deque is either empty or contains only a single node.

b) The number of DCAS calls is 1 per uncontended push or pop operation.

c) A full storage width DCAS primitive that operates on two independently-addressable storage units may be employed. Accordingly, full storage width is available for addresses or data and tag bits need not be set aside.

d) The set of values that may be pushed onto the deque is not limited by use of special values to mark nodes of the linked-list.

e) Explicit reclamation may be provided in some realizations.

Although all of these features is provided in some realizations, fewer than all may be provided in others.

The organization and structure of a doubly-linked-list and deque encoded therein are now described with reference to **FIGS. 1A, 1B, 1C** and **1D**. In general, individual elements of the linked-list can be represented as instances of a simple node structure. For example, in one realization, nodes are implemented in accordance with the following definition:

```
typedef Node {
  Node *R;
  Node *L;
  val V;
}

Node *LeftHat = null;
Node *RightHat = null;
```

Each node encodes two link pointers, `R` and `L`, and a value, `V`. There are two global "anchor" variables, or hats, called `LeftHat` and `RightHat`, which generally point to the leftmost node and the rightmost node in the doubly linked chain.

A linked-list **101A** of such nodes is illustrated in **FIG. 1A**. Linked-list **101A** encodes a deque **102** with a state corresponding to the sequence, $S = \langle A, B, C, D, \ldots, Y, Z \rangle$. Elements of the sequence are encoded as values in `V` fields of respective nodes of linked-list **101A**. In general, the value field of the illustrated structure may include a literal value or a pointer value. Particular data structures identified by pointer values are, in general, application specific. Literal values may be appropriate for some applications, while in others, more complex node structures may be employed. Based on the description herein, these and other variations will be appreciated by persons of ordinary skill in the art. Nonetheless, and without loss of generality, the simple node structure defined above is used for purposes of illustration.

As a general rule, the `V` field of the node pointed to by `LeftHat` is the leftmost item in the deque and the `V` field of the node pointed to by `RightHat` is the rightmost item in the deque. If both hats contain null pointers, then the deque is empty. Contents of the `L` field of the node to which `LeftHat` points generally do not matter, e.g., the `L` field of such a node may be null or may point to another node. However, we have a special rule, whose significance will be understood in the context of the description that follows, that if `LeftHat` points to a node whose `L` field points to that same node (in which case we say that the `L` field contains a self-pointer), then the queue is logically empty and the `V` field of that node is not an item in the deque. Similarly, contents of the `R` field of the node to which `RightHat` points generally do not matter, e.g., the `R` field of such a node may be null or may point to another node. Again, we have the special rule, that if `RightHat` points to a node whose `R` field points to that same node (in which case we say that the `R` field contains a self-pointer), then the queue is logically empty and the `V` field of that node is not an item in the deque. Once an `L` field or an `R` field contains a self-pointer, that field is not changed again (until the node is determined to be inaccessible and therefore eligible to be reclaimed).

FIGS. 1A, 1B, 1C and **1D** illustrate various alternate encodings of deque **102** in accordance with the above and which will be better understood in the context of push and pop operation descriptions that follow.

In particular, **FIG. 1A** illustrates a linked-list **101A** state encoding deque **102** after successful push operations at each end thereof. **FIG. 1B** illustrates a linked-list **101B** state encoding deque **102** after successful pop operations at each end thereof. **FIG. 1C** illustrates a linked-list **101C** state encoding deque **102** after successful left pop and right push operations. **FIG. 1D** illustrates a linked-list **101D** state encoding deque **102** after successful left push and right pop operations.

FIGS. 2A, 2B, 2C and **2D** illustrate another set of corresponding linked-list and deque states, namely those corresponding to an empty deque. One encoding of an empty deque, illustrated in **FIG. 2A**, is when `LeftHat` and `RightHat` are both `null`. However, in addition, several other states in which either or both of `LeftHat` and `RightHat` are non-`null` are also treated as logically empty deque encodings. **FIGS. 2B, 2C** and **2D** illustrate such additional logically empty deque encodings. In the particular encodings illustrated, either a `null`-valued hat (`LeftHat` and `RightHat`) or a hat that points to a node with a corresponding self-pointer (e.g., `LeftHat` pointing to a node with a left pointer that is a self-pointer or `RightHat` pointing to a node with a right pointer that is a self-pointer) is indicative of a logically empty deque. Another encoding of an empty deque state wherein both the `LeftHat` and `RightHat` point to a marker or "Dummy" node is described in greater detail below.

Access operations

The description that follows presents an exemplary non-blocking implementation of a deque based on an underlying doubly-linked-list data structure wherein access operations (illustratively, `push_right`, `pop_right`, `push_left` and `pop_left`) facilitate concurrent access. Exemplary code and illustrative drawings will provide persons of ordinary skill in the art with detailed understanding of one particular realization of the present invention; however, as will be apparent from the description herein and the breadth of the claims that follow, the invention is not limited thereto. Exemplary right-hand-side code is described in detail with the understanding that left-hand-side operations are symmetric. Use herein of directional signals (e.g., left and right) will be understood by persons of ordinary skill in the art to be somewhat arbitrary. Accordingly, many other notational conventions, such as top and bottom, first-end and second-end, etc., and implementations denominated therein are also suitable.

An illustrative `push_right` access operation in accordance with the present invention follows:

```
val push_right(val v) {
  nd = new Node();  /* Allocate new Node structure */
  if (nd == null) return "full";
  nd->R = null;
  nd->V = v;
  while (true) {
    rh = RightHat;
    if (rh == null || ((rhR = rh->R) == rh)) {
      nd->L = null;
      lh = LeftHat;
      if (DCAS(&RightHat, &LeftHat, rh, lh, nd, nd))
        return "okay";
    }
    else {
```

```
        nd->L = rh;
        if (DCAS(&RightHat, &rh->R, rh, rhR, nd, nd))
          return "okay";
      }
    }
  }
```

The `push_right` operation first obtains a fresh Node structure from the storage allocator (line 2). We assume that if allocatable storage has been exhausted, the new operation will yield a `null` pointer. The `push_right` operation treats this as sufficient cause to report that the deque is full (line 3). Otherwise, the `R` field of the new node is set to null (line 4) and the value to be pushed is stored into the `V` field (line 5). All that remains is to splice this new node into the doubly linked chain. However, an attempt to splice may fail (because of an action taken by some other concurrent push or pop operation). Accordingly, a "while true" loop (line 6) is used to iterate until a splicing attempt succeeds.

The `RightHat` is copied into local variable `rh` (line 7). If `rh` is `null` or points to a node whose `R` field contains a self-pointer (line 8), then the deque is logically empty. In this case, the new node should become the only node in the deque. Its `L` field is set to null (line 9) and then a DCAS is used (line 11) to atomically make both `RightHat` and `LeftHat` point to the new node—but only if neither `RightHat` nor `LeftHat` has changed. If this DCAS operation succeeds, then the push has succeeded (line 12). If the DCAS fails, then operation of the "while true" loop will cause a retry.

If the deque is not logically empty, then the new node should be added to the right-hand end of the doubly linked chain. The copied content of the `RightHat` (stored locally in `rh`) is stored into the `L` field of the new node (line 15) and then a DCAS is used (line 16) to make both the `RightHat` and the former right-end node point to the new node, which thus becomes the new right-end node. If this DCAS operation succeeds, then the push has succeeded (line 17). If the DCAS fails, then operation of the "while true" loop will cause a retry.

An illustrative `pop_right` access operation in accordance with the present invention follows:

```
val pop_right() {
  while (true) {
    rh = RightHat;
    lh = LeftHat;
    if (rh == null) return "empty";
    if (rh->R == rh) {
     if (DCAS(&RightHat, &rh->R, rh, rh, rh, rh)
       return "empty";
     else
       continue;  // go back to top of loop
    }
    if (rh == lh) {
      if (DCAS(&RightHat, &LeftHat, rh, lh, null, null))
        return rh->V;
    }
    else {
      rhL = rh->L;
      if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) {
```

```
                result = rh->V;
                rh->R = null;
                rh->V = null;
                return result;
            }
        }
    }
}
```

The right-side pop operation also uses a "while true" loop (line 2) to iterate until an attempt to pop succeeds. The `RightHat` is copied into local variable `rh` (line 3). If `rh` is `null` or points to a node whose `R` field contains a self-pointer, then the deque is logically empty and the pop operation reports that fact (line 5, line 8). Because changes are possible between the time we read the `RightHat` and the time we read the `R` pointer, we use a DCAS (line 7) to verify that these two pointers, are at the same moment, equal to the values individually read. If the deque is non-empty, there are two cases, depending on whether there is one element or more than one element in the deque. There is one element in the deque only if the `LeftHat` and `RightHat` point to the same node (line 12). As with the previously illustrated states, a variety of linked-list states may encode a one-element deque state. **FIGS. 3A, 3B, 3C** and **3D** illustrate such one-element deque states. In the case of a one-element deque state, a DCAS operation is used to reset both hats to `null` (line 13). If the DCAS succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node (line 14). In the realization above, it is assumed that, after completion of the `pop_right` operation, the node just popped will be reclaimed by an automatic storage reclamation facility, using garbage collection or some such technique. Another realization (described below) allows for explicit reclamation.

If there is more than one element in the deque, then the rightmost node should be removed from the doubly linked chain. A DCAS is used (line 18) to move the `RightHat` to the node to the immediate left of the rightmost node and to change the `L` field of that rightmost node to contain a self-pointer. A subtle point is that, in the above realization, the variable `rhL`, assigned in line 17, never contains a null pointer, even if nodes have been popped concurrently from the left-hand end of the deque. Accordingly, the DCAS in line 18 never changes `RightHat` to be a null pointer.

If the DCAS (line 18) fails, then operation of the "while true" loop will cause a retry. If the DCAS succeeds, then the pop succeeds and the value to be returned is in the `V` field of the popped node. Before this value is returned, the `R` field and the `V` field of the popped node are cleared. It is important to clear the `R` field (line 20) so that previously popped nodes may be reclaimed. It may be also desirable to clear the `V` field immediately (line 21) so that the popped value, if used quickly and then discarded by the caller, will not be retained indefinitely. If the `V` field does not contain a reference to other data structures, then line 21 may be omitted.

As described above, left variants of the above-described right push and pop operations are symmetric. However, for clarity, each is presented below. An illustrative `push_left` access operation in accordance with the present invention follows:

```
val push_left(val v) {
  nd = new Node();  /* Allocate new Node structure */
  if (nd == null) return "full";
  nd->L = null;
  nd->V = v;
  while (true) {
    lh = LeftHat;
    if (lh == null || ((lhL = lh->L) == lh)) {
      nd->R = null;
      rh = RightHat;
      if (DCAS(&LeftHat, &RightHat, lh, rh, nd, nd))
        return "okay";
    }
    else {
      nd->R = lh;
      if (DCAS(&LeftHat, &lh->L, lh, lhL, nd, nd))
        return "okay";
    }
  }
}
```

An illustrative `pop_left` access operation in accordance with the present invention follows:

```
val pop_left() {
  while (true) {
    lh = LeftHat;
    rh = RightHat;
    if (lh == null) return "empty";
    if (lh->L == lh) {
      if(DCAS(&LeftHat, &lh->L, lh, lh, lh, lh))
        return "empty";
      else
        continue;  // go back to top of loop
    }
    if (lh == rh) {
      if (DCAS(&LeftHat, &RightHat, lh, rh, null, null))
        return lh->V;
    }
    else {
      lhR = lh->R;
      if (DCAS(&LeftHat, &lh->R, lh, lhR, lhR, lh)) {
        result = lh->V;
        lh->L = null;
        lh->V = null;
        return result;
      }
    }
  }
}
```

The push and pop operations work together in a relatively straightforward manner except in one odd case. If a `pop_right` operation and a `pop_left` operation occur concurrently when there are exactly two nodes in the deque, e.g., as illustrated in **FIGS. 4A, 4B, 4C** and **4D**, then each operation may (correctly) discover that `LeftHat` and `RightHat` point to different nodes (see line 12 in the `pop_right` and `pop_left` realizations above) and therefore proceed to perform a DCAS for the multinode case (line 18 the `pop_right` and `pop_left` realizations above). Both of these DCAS operations may succeed, because they operate on

disjoint pairs of memory locations. The result is that the hats pass each other as illustrated by the logically empty state of **FIG. 2B**.

Because there had been two nodes in the deque and both have been popped, the deque should be regarded as logically empty. The self-pointers encoded in respective left and right pointers of nodes identified by `LeftHat` and `RightHat` by operation of the `pop_left` and `pop_right` described above serve to identify the logically empty state to subsequent push and pop operations (see line 8 of `push_right` and `push_left` and lines 6 and 7 of `pop_right` and `pop_left`).

Self-Pointer Alternatives

Although self-pointers have been used throughout the access operations described above, persons of ordinary skill in the art will recognize that other distinguishing pointer values are also suitable. In general, it is suitable to compare appropriate list pointers (e.g., `lh->L` for left-end access operations and `rh->R` for right-end access operations) with any distinguishing value that is guaranteed not to be employed in an interior list pointer. In short, use of self-pointers is convenient though not required.

For example, in one realization in accordance with the present invention, a "marker node" may be defined as follows:

```
Node Dummy = new Node();
```

In such a realization, the assignments of self-pointers and predicates that check for self-pointers in the above-described access operations may be replaced with assignments (e.g., by DCAS) of a pointer to the marker node and with predicates that check for such a marker node pointer.

Another improvement can be made by avoiding the use of the `null` encoding in the outward pointer(s) of a newly added (i.e., pushed) node. For example, in one realization in accordance with the present invention, a "marker node" may be defined as follows:

```
Node Dummy = new Node();
Dummy.R = Dummy;
Dummy.L = Dummy;
```

Then, let every occurrence of "`null`" in the above access operations (`push_right`, `pop_right`, `push_left` and `pop_left`), except those in line 3 of the `push_right` and `push_left` operations, be replaced with "Dummy". Then, the tests represented by the left-hand operands of all occurrences of `||` become redundant and may be removed (together with the `||` operators themselves). The modified `push_right` access operation:

```
val push_right(val v) {
  nd = new Node();  /* Allocate new Node structure */
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
```

```
      rh = RightHat;
      if ((rhR = rh->R) == rh)) {
        nd->L = Dummy;
        lh = LeftHat;
        if (DCAS(&RightHat,&LeftHat,rh,lh,nd,nd))
          return "okay";
      }
      else {
        nd->L = rh;
        if (DCAS(&RightHat,&rh->R,rh,rhR,nd,nd))
          return "okay";
      }
    }
  }
```

and pop_right access operation:

```
  val pop_right() {
    while (true) {
      rh = RightHat;
      lh = LeftHat;
      if (rh->R == rh) {
       if (DCAS(&RightHat,&rh->R,rh,rh,rh,rh)
         return "empty";
       else
         continue;  // go back to top of loop
      }
      if (rh == lh) {
        if (DCAS(&RightHat,&LeftHat,rh,lh,Dummy,Dummy))
          return rh->V;
      }
      else {
        rhL = rh->L;
        if (DCAS(&RightHat,&rh->L,rh,rhL,rhL,rh)) {
          result = rh->V;
          rh->R = Dummy;
          rh->V = null;
          return result;
        }
      }
    }
  }
```

are illustrative. Left variants of the above-described left push and pop operations are symmetric.

Use of a marker node and elimination of the `null` encoding in the outward pointer(s) of a newly added node allows a further variation, i.e., use of a `null` value in place of a self pointer, as a distinguishing pointer value. For example, a "marker node" may be defined as follows:

```
Node Dummy = new Node();
Dummy.R = null;
Dummy.L = null;
```

By making a corresponding modification to the `push_right` and `pop_right` access operations, self-pointers can be fully replaced with use of a `null` value as a distinguishing pointer value. For example, the `push_right` access operation may be rewritten as follows:

```
val push_right(val v) {
  nd = new Node();  /* Allocate new Node structure */
  if (nd == null) return "full";
  nd->R = Dummy;
  nd->V = v;
  while (true) {
    rh = RightHat;
    if ((rhR = rh->R) == null)) {
      nd->L = Dummy;
      lh = LeftHat;
      if (DCAS(&RightHat,&LeftHat,rh,lh,nd,nd))
        return "okay";
    }
    else {
      nd->L = rh;
      if (DCAS(&RightHat,&rh->R,rh,rhR,nd,nd))
        return "okay";
    }
  }
}
```

where the predicate in line 8 is modified to instead check for a `null` value as the distinguishing pointer value.

The `pop_right` access operation is similarly modified:

```
val pop_right() {
  while (true) {
    rh = RightHat;
    lh = LeftHat;
    if (rh->R == null) {
     if (DCAS(&RightHat,&rh->R,rh,null,rh,null)
       return "empty";
     else
       continue;  // go back to top of loop
    }
    if (rh == lh) {
      if (DCAS(&RightHat,&LeftHat,rh,lh,Dummy,Dummy))
        return rh->V;
    }
    else {
      rhL = rh->L;
      if (rhL != null &&
          DCAS(&RightHat,&rh->L,rh,rhL,rhL,null)) {
        result = rh->V;
        rh->R = null;
        rh->V = null;
        return result;
      }
    }
  }
}
```

where the predicate in line 5 is modified to instead check for a `null` value and the final operand of the DCAS operation in line 13 is modified to instead install a `null` value as the distinguishing pointer value. Because we have eliminated the self-pointers, it is possible to see a `null` value in `rh->L`. We avoid dereferencing in this case and instead retry. Elimination of self-pointers may have some benefit in some realizations in which

the elimination of cycles facilitates reclamation of storage. One such explicit reclamation realization is described below.

Explicit Reclamation of Storage

While the above description has focused on implementations for execution environments that provide intrinsic support for automatic reclamation of storage, or garbage collection, some implementations in accordance with the present invention support explicit reclamation. This is important for several reasons. First, many common programming environments do not support garbage collection. Second, almost all of those that do provide garbage collection introduce excessive levels of synchronization overhead, such as locking and/or stop-the-world collection mechanisms. Accordingly, the scaling of such implementations is questionable. Finally, designs and implementations that depend on existence of a garbage collector cannot be used in the implementation of the garbage collector itself.

It has been discovered that a variation on the above-described techniques may be employed to provide explicit reclamation of nodes as they are severed from the deque as a result of pop access operations. The variation builds on a lock-free reference counting technique that allows us to transform a garbage-collection-dependent concurrent data structure implementation that satisfies the two criteria into an equivalent implementation that does not depend on garbage collection. These criteria are:

1. **LFRC Compliance** The implementation does not access or manipulate pointers other than through a set of pointer operations that ensure that if the number of pointers to an object is non-zero, then so too is its reference count, and that if the number of pointers is zero, then the reference count eventually becomes zero. For example, compliance with such a criterion generally precludes the use of pointer arithmetic, unless the implementation thereof enforces the criterion. For example, in some implementations, arithmetic operations on pointers could be overloaded with compliant versions of the arithmetic operations. In an illustrative realization described below, an implementation of a concurrent shared object accesses and manipulates pointers only through a set of functions, procedures or methods (e.g., load, store, copy, destroy, CAS and/or DCAS operations) that ensure compliance.

2. **Cycle-Free Garbage** There are no pointer cycles in garbage. Note that, cycles may exist in the concurrent data structure, but not amongst objects that have been removed from the data structure, and which should be freed.

Our transformation preserves lock-freedom. In particular, if the original implementation is lock-free, so too is the garbage-collection-independent algorithm.

LFRC Operations – An Illustrative Set

An illustrative set of LFRC pointer operations is now described. As stated above, we assume that pointers in a data structure implementation under consideration are accessed *only* by means of these operations.

1. `LFRCLoad(A,p)` — `A` is a pointer to a shared memory location that contains a pointer, and `p` is a pointer to a local pointer variable. The effect is to load the value from the location pointed to by `A` into the variable pointed to by `p`.

2. `LFRCStore(A,v)` — `A` is a pointer to a shared memory location that contains a pointer, and `v` is a pointer value to be stored in this location.

3. `LFRCCopy(p,v)` — `p` is a pointer to a local pointer variable and `v` is a pointer value to be copied to the variable pointed to by `p`.

4. `LFRCDestroy(v)` — `v` is the value of a local pointer variable that is about to be destroyed.

5. `LFRCCAS(A0,old0,new0)` — `A0` is a pointer to a shared memory location that contains a pointer, and `old0` and `new0` are pointer values. The effect is to atomically compare the contents of the location pointed to by `A0` with `old0` and to change these contents to `new0` and return *true* if the comparison succeeds; if it fails, then the contents of the location pointed to by `A0` are left unchanged, and `LFRCCAS` returns *false*.

6. `LFRCDCAS(A0,A1,old0,old1,new0,new1)` — `A0` and `A1` are pointers to shared memory locations that contain pointers, and `old0`, `old1`, `new0`, and `new1` are pointer values. The effect is to atomically compare the contents of the location pointed to by `A0` with `old0` and the contents of the location pointed to by `A1` with `old1`, to change the contents of the locations pointed to by `A0` and `A1` to `new0` and `new1`, respectively, and to return `true` if the comparisons both succeed; if either comparison fails, then the contents of the locations pointed to by `A0` and `A1` are left unchanged, and `LFRCDCAS` returns *false*.

FIG. 5 depicts a shared memory multiprocessor configuration in which the illustrated set of LFRC pointer operations may be employed. In particular, **FIG. 5** depicts a pair of processors **511** and **512** that access storage **540**. Storage **540** includes a shared storage portion **530** and local storage portions **521** and **522**, respectively accessible by execution threads executing on processors **511** and **512**. In general, the multiprocessor configuration is illustrative of a wide variety of physical implementations, including implementations in which the illustrated shared and local storage portions correspond to one or more underlying physical structures (e.g., memory, register or other storage), which may be shared, distributed or partially shared and partially distributed.

Accordingly, the illustration of **FIG. 5** is meant to exemplify an architectural view of a multiprocessor configuration from the perspective of execution threads, rather than any particular physical implementation. Indeed, in some realizations, data structures encoded in shared storage portion **530** (or portions thereof) and local storage (e.g., portion **521** and/or **522**) may reside in or on the same physical structures. Similarly, shared storage portion **530** need not correspond to a single physical structure. Instead, shared storage portion **530** may correspond to a collection of sub-portions each associated with a processor, wherein the multiprocessor configuration provides communication mechanisms (e.g., message passing facilities, bus protocols, etc.) to architecturally present the collection of sub-portions as shared storage.

Furthermore, local storage portions **521** and **522** may correspond to one or more underlying physical structures including addressable memory, register, stack or other storage that are architecturally presented as local to a corresponding processor. Persons of ordinary skill in the art will appreciate a wide variety of suitable physical implementations whereby an architectural abstraction of shared memory is provided. Realizations in accordance with the present invention may employ any such suitable physical implementation.

In view of the foregoing and without limitation on the range of underlying physical implementations of the shared memory abstraction, LFRC pointer operations may be better understood as follows. Pointer `A` references a shared memory location **531** that contains a pointer to an object **532** in shared memory. One or more pointers such as pointer `A` is (are) employed as operands of the `LFRCLoad`, `LFRCStore`, `LFRCCAS` and `LFRCDCAS` operations described herein. Similarly, pointer `p` references local storage **534** that contains a pointer to an object (e.g., object **532**) in shared memory. In this regard, **FIG. 5** illustrates a state, `*A == *p`, consistent with successful completion of either a `LFRCLoad` or `LFRCStore` operation. In general, pointers `A` and `p` may reside in any of a variety storage locations. Often, both pointers reside in storage local to a particular processor. However, either or both of the pointers may reside elsewhere, such as in shared storage.

In our experience, the operations presented above are typically sufficient for many concurrent shared object implementations, but can result in somewhat non-transparent code. Accordingly, we have also implemented some extensions that allow more elegant programming and handle issues such as the pointer created by passing a pointer by value transparently. For example,

1. `p = LFRCLoad2 (A)` — `A` is a pointer to a shared memory location that contains a pointer, and `p` is a local pointer variable, where `p` is known not to contain a pointer (e.g., it has just been declared). The effect is to load the value from the location pointed to by `A` into `p`.

2. `LFRCStoreAlloc (A, v)` — `A` is a pointer to a shared memory location that contains a pointer, and `v` is a pointer value that will not be used (or destroyed) again. Accordingly, there is no need to increment a reference count corresponding to `v`. This variation is useful when we want to invoke an allocation routine directly as the second parameter, e.g., as `LFRCStoreAlloc(&X,allocate_structure())`.

3. `LFRCDCAS2(A0,A1,old0,old1,new0,new1)` — `A0` is a pointer to a shared memory location that contains a pointer, `A1` is a pointer to a shared memory location that contains a non-pointer value, `old0` and `new0` are pointer values, and `old1` and `new1` are values, e.g., literals, for which no reference counting is required.

4. `LFRCPass (p)` — `p` is a pointer value to be passed by value and for which a reference count should be incremented. This variation is useful when we want to pass `p` to a routine, e.g., as `Example( , , …,LFRCPass(p))`.

Based on the description herein, persons of ordinary skill in the art will appreciate variations of the described implementations, which may employ these and other extensions and/or variations on a set of supported pointer operations.

LFRC Transformation

Building on the previously described illustrative set of pointer operations, we transform from a GC-dependent implementation into a GC-independent implementation as follows.

1. **Add reference counts:** Add a reference count field `rc` to each object type to be used by the implementation. This field should be set to 1 in a newly-created object (in an object-oriented language such as C++, initialization may be achieved through object constructors).

2. **Provide an `LFRCDestroy(v)` function:** Write a function `LFRCDestroy(v)` that accepts a pointer `v` to an object. If `v` is `NULL`, then the function should simply return; otherwise it should atomically decrement `v->rc`. If the reference count field becomes zero as a result, `LFRCDestroy` should recursively call itself with each pointer in the object, and then free the object. An example is provided below, and we provide a function (`add_to_rc`) for the atomic decrement of the `rc` field, so writing this function is straightforward. We employ this function only because it is the most convenient and language-independent way to iterate over all pointers in an object. Other implementations may provide similar facility using language-specific constructs.

3. **Ensure no garbage cycles:** Ensure that the implementation does not result in referencing cycles in or among garbage objects. Note that, as illustrated below, the concurrent data structure may include cycles. However, storage no longer reachable should not include cycles.

4. **Produce correctly-typed LFRC pointer operations:** We have provided code for the LFRC pointer operations to be used in the example implementation presented in the next section. In this implementation, there is only one type of pointer. For simplicity, we have explicitly designed the LFRC pointer operations for this type. For other simple concurrent shared object implementations, this step can be achieved by simply replacing the `Node` type used in this implementation with a new type. In algorithms and data structure implementations that use multiple pointer types, a variety of alternative implementations are possible. In general, operations may be duplicated for the various pointer types or, alternatively, the code for the operations may be unified to accept different pointer types. For example, in some realizations an `rc` field may be defined uniformly, e.g., at the same offset in all objects, and `void` pointers may be employed instead of specifically-types ones. In such realizations, definition of multiple object-type-specific LFRC pointer operations can be eliminated. Nonetheless, for clarity of illustration, an object-type-specific set of LFRC pointer operations is described below.

5. **Replace pointer operations:** Replace each pointer operation with its LFRC pointer operation counterpart. For example, if `A0` and `A1` are pointers to shared pointer variables, and `x0`, `x1`, `old0`, `old1`, `new0`, `new1` are pointer variables, then replacements may be made as follows:

Replaced Pointer Operation	LFRC Pointer Operation
`x0 = *A0;`	`LFRCLoad(A0, &x0);`
`*A0 = x0;`	`LFRCStore(A0,x0);`
`x0 = x1;`	`LFRCCopy(&x0,x1);`
`CAS(A0,old0,new0)`	`LFRCCAS(A0,old0,new0)`
`DCAS(A0,A1,old0,old1,new0,new1)`	`LFRCCAS(A0,A1,old0,old1,new0,new1)`

Note that the table does not contain an entry for replacing an assignment of one shared pointer value to another, for example `*A0=*A1`. Such assignments are not atomic. Instead, the location pointed to by `A1` is read into a register in one instruction, and the contents of the register are stored into the location pointed to by `A0` in a separate instruction. This approach should be reflected explicitly in a transformed implementation, e.g., with the following code:

```
{
    ObjectType *x = NULL;
    LFRCLoad(A1, &x);
    LFRCStore(A0,x);
    LFRCDestroy(x);
}
```

or its substantial equivalent, whether included directly or using a "wrapper" function.

6. **Management of local pointer variables:** Finally, whenever a thread loses a pointer (e.g., when a function that has local pointer variables returns, so its local variables go out of scope), it first calls `LFRCDestroy()` with this pointer. In addition, pointer variables are initialized to `NULL` before being used with any of the LFRC operations. Thus, pointers in a newly-allocated object should be initialized to `NULL` before the object is made visible to other threads. As illustrated in the example below, it is also important to explicitly remove pointers contained in a statically allocated object before destroying that object.

Explicitly Reclaimed Concurrent Deque Implementation

As before, the deque is represented as a doubly-linked list of nodes, the leftmost and rightmost nodes in a non-empty deque are identified by `LeftHat` and `RightHat`, respectively, and a "marker node," `Dummy`, with `null` value left and right pointers, is used as a sentinel node at one or both ends of the deque. On initialization, both `LeftHat` and `RightHat` are set to point to the marker node. To facilitate explicit reclamation, `null` pointers (rather than self-pointers) are used to as distinguishing pointer values so that reference cycles are avoided.

Since the `null` pointer implementation ensures that referencing cycles do not exist in or among nodes severed from the list, the implementation described is amenable to transformation to a GC-independent form using the lock-free reference counting (LFRC) methodology described above. However, to summarize, (1) we added a reference count field `rc` to the node object, (2) we implemented an `LFRCDestroy(v)` function, (3) we ensured (using the `null` pointer implementation) that the implementation does not result in referencing cycles in or among garbage objects, (4, 5) we replaced accesses and manipulations of pointer variables with corresponding LFRC pointer operations and (6) we ensured that local pointer variables are

initialized to NULL before being used with any of the LFRC operations and are properly destroyed using LFRCDestroy upon return (or when such local pointer variables otherwise go out of scope). LFRC pointer operations employed include LFRCLoad, LFRCStore, LFRCDCAS and LFRCDestroy. An illustrative implementation of each is described below.

An illustrative push_right access operation in accordance with the present invention follows:

```
val push_right(val v) {
  nd = new Node();  /* Allocate new Node structure */
  rh = null; rhR = null; lh = null;
  if (nd == null) {
    LFRCDestroy(lh,rhR,rh,nd);
    return "full";
  }
  LFRCStore(&nd->R,Dummy);
  nd->V = v;
  while (true) {
    LFRCLoad(&RightHat,&rh);
    LFRCLoad(&rh->R,&rhR);
    if (rhR == null) {
      LFRCStore(&nd->L,Dummy);
      LFRCLoad(&LeftHat,&lh);
      if (LFRCDCAS(&RightHat,&LeftHat,rh,lh,nd,nd)) {
        LFRCDestroy(lh,rhR,rh,nd);
        return "okay";
      }
    } else {
      LFRCStore(&nd->L,rh);
      if (LFRCDCAS(&RightHat,&rh->R,rh,rhR,nd,nd)) {
        LFRCDestroy(lh,rhR,rh,nd);
        return "okay";
      }
    }
  }
}
```

where the notation LFRCDestroy(lh,rhR,rh,nd) is shorthand for operation of the LFRCDestroy operation on each of the listed operands.

An illustrative pop_right access operation in accordance with the present invention follows:

```
val pop_right() {
  rh = null; lh = null; rhL = null; tmp = null;
  while (true) {
    LFRCLoad(&RightHat,&rh);
    LFRCLoad(&LeftHat,&lh);
    LFRCLoad(&rh->R,&tmp);
    if (tmp == null)
      if (LFRCDCAS(&RightHat,&rh->R,rh,null,rh,null)) {
        LFRCDestroy(tmp,rhL,lh,rh);
        return "empty";
      } else continue;
    if (rh == lh) {
      if (LFRCDCAS(&RightHat,&LeftHat,rh,lh,Dummy,Dummy)) {
        result = rh->V;
        LFRCDestroy(tmp,rhL,lh,rh);
```

```
            return result;
          }
        } else {
          LFRCLoad(&rh->L, &rhL);
          if (rhL != null &&
              LFRCDCAS(&RightHat, &rh->L, rh, rhL, rhL, null)) {
            result = rh->V;
            LFRCStore(&rh->R, null);
            LFRCDestroy(tmp, rhL, lh, rh);
            return result;
          }
        }
      }
    }
```

wherein the LFRCDCAS pointer operation employed (at line 8) to ensure proper empty deque detection may optionally remain a DCAS primitive (as illustrated in the GC-dependent implementation). A DCAS primitive without LFRC support is possible because, whether or not successful, the DCAS does not alter any referencing state. Nonetheless, exemplary code above illustrates the LFRCDCAS pointer operation because some implementations may prefer to forgo such an optimization in favor of a simpler transformation from GC-dependent to explicitly-reclaimed form.

As before, it is possible to see a `null` value in `rh->L` and we avoid dereferencing in this case. Since this `null` value can only appear if the deque state changes while the `pop_right` operation is executing, we can safely retry without compromising lock-freedom. As described above, left variants of the above-described right push and pop operations are symmetric.

Implementation of LFRC Pointer Operations

In the description that follows, we describe an illustrative implementation of LFRC pointer operations and explain why the illustrative operations ensure that there are no memory leaks and that memory is not freed prematurely. The LFRC pointer operations maintain a reference count in each object, which reflects the number of pointers to the object. When this count reaches zero, there are no more pointers to the object and the object can be freed.

The main difficulty is that we cannot atomically change a pointer variable from pointing to one object to pointing to another and update the reference counts of both objects. We overcome this problem with the observations that:

1. provided an object's reference counter is always *at least* the number of pointers to the object, it will never be freed prematurely, and

2. provided the count eventually becomes zero when there are no longer any pointers to the object, there are no memory leaks.

Thus, we conservatively increment an object's reference count *before* creating a new pointer to it. If we subsequently fail to create that pointer, then we can decrement the reference count again afterwards to

reflect that the new pointer was not created. An important mechanism in the illustrated implementation is the use of DCAS to increment an object's reference count while simultaneously checking that some pointer to the object exists. This avoids the possibility of updating an object after it has been freed, thereby potentially corrupting data in the heap, or in an object that has been reallocated.

We now describe a lock-free implementation of the LFRC pointer operations, beginning with an implementation of `LFRCLoad` as follows:

```
void LFRCLoad(SNode **A,  SNode **dest) {
      SNode *a, *olddest = *dest;
      long r;
      while (true) {
            a = *A;
            if (a == Null) {
                  *dest = Null;
                  break;
            }
            r = a->rc;
            if (DCAS(A, &a->rc, a, r, a, r+1)) {
                  *dest = a;
                  break;
            }
      }
      LFRCDestroy(olddest);
}
```

where `LFRCLoad` accepts two parameters, a pointer `A` to a shared pointer, and a pointer `dest` to a local pointer variable of the calling thread. The semantics of the `LFRCLoad` operation is to load the value in the location pointed to by `A` into the variable pointed to by `dest`. This has the effect of destroying one pointer (the previous value in the location pointed to by `dest`) and creating another (the new value of `*dest`). Thus, we must potentially update two reference counts. The `LFRCLoad` operation begins by recording the previous value of the pointer (line 2), so that it can be destroyed later. Note that we cannot destroy it yet, as this would risk destroying the object to which we are about to create a pointer.

Next, the `LFRCLoad` operation loads a new value from `*A` and, if the pointer read is non-`NULL`, increments the reference count of the object pointed to by `*A` in order to record that a new pointer to this object has been created. In this case, because the calling thread does not (necessarily) already have a pointer to this object, it is not safe to update the reference count using a simple CAS primitive. The object might be freed before the CAS executes, creating a risk that execution of the CAS modifies a location in a freed object or in an object that has subsequently been reallocated for another purpose. Therefore, the `LFRCLoad` operation uses a DCAS primitive to attempt to atomically increment the reference count, while ensuring that the pointer to the object still exists.

In the above implementation, these goals are achieved as follows. First, the `LFRCLoad` operation reads the contents of `*A` (line 5). If it sees a `NULL` pointer, there is no reference count to be incremented, so `LFRCLoad` simply sets `*dest` to `NULL` (lines 6-8). Otherwise, it reads the current reference count of the object pointed to by the pointer it read in line 5, and then attempts to increment this count using a DCAS (line

11) to ensure that the pointer to the object containing the reference count still exists. Note that there is no risk that the object containing the pointer being read by LFRCLoad is freed during the execution of LFRCLoad because the calling thread has a pointer to this object that is not destroyed during the execution of LFRCLoad. Accordingly, the reference count cannot fall to zero. If the DCAS succeeds, then the value read is stored (line 12) in the variable passed to LFRCLoad for this purpose. Otherwise, LFRCLoad retries. After LFRCLoad succeeds in either loading a NULL pointer, or loading a non-NULL pointer and incrementing the reference count of the object to which it points, it calls LFRCDestroy in order to record that the pointer previously in *dest has been destroyed (line 16).

An illustrative implementation of the LFRCDestroy operation will be understood as follows:

```
10  void LFRCDestroy(SNode *p) {
        if (p != Null && add_to_rc(p, -1)==1) {
            LFRCDestroy(p->L, p->R);
            delete p;
        }
15  }
```

If the LFRCDestroy operation's argument is non-NULL, then it decrements the reference count of the object pointed to by its argument (line 2, above). This is done using an add_to_rc function (such as that shown below) implemented using a CAS primitive. The add_to_rc function is safe (in the sense that there is no risk that it will modify a freed object) because it is called only in situations in which we know that the calling thread has a pointer to this object, which has previously been included in the reference count. Therefore, there is no risk that the reference count will become zero, thereby causing the object to be freed, before the add_to_rc function completes. If execution of the add_to_rc function causes the reference count to become zero, then we are destroying the last pointer to this object, so it can be freed (line 4, above). First, however, LFRCDestroy calls itself recursively (line 3, above) with each pointer in the object in order to update the reference counts of objects to which the soon-to-be-freed object has pointers.

```
long add_to_rc(SNode *p,  int v) {
    long oldrc;
    while (true) {
        oldrc = p->rc;
        if (CAS(&p->rc), oldrc, oldrc+v)
            return oldrc;
    }
}
```

An LFRCStore operation can be implemented as follows:

```
void LFRCStore(SNode **A, SNode *v) {
    SNode *oldval;
    if (v != Null)
        add_to_rc(v, 1);
    while (true) {
        oldval = *A;
        if (CAS(A, oldval, v)) {
            LFRCDestroy(oldval);
            return;
```

```
            }
        }
}
```

where the LFRCStore operation accepts two parameters, a pointer A to a location that contains a pointer, and a pointer value v to be stored in this location. If the value v is not NULL, then the LFRCStore operation increments the reference count of the object to which v points (lines 3-4). Note that at this point, the new pointer to this object has not been created, so the reference count is greater than the number of pointers to the object. However, this situation will not persist past the end of the execution of the LFRCStore operation, since LFRCStore does not return until that pointer has been created. In the illustrated implementation, the pointer is created by repeatedly reading the current value of the pointer and using a CAS primitive to attempt to change the contents of the location referenced by A to the pointer value v (lines 5-9). When the CAS succeeds, we have created the pointer previously counted and we have also destroyed a pointer, namely the previous contents of *A. Therefore, LFRCStore calls LFRCDestroy (line 8) to decrement the reference count of the object to which the now-destroyed pointer points.

Finally, a LFRCDCAS operation can be implemented as follows:

```
bool LFRCDCAS(SNode **A0,  SNode **A1,
              SNode *old0,  SNode *old1,
              SNode *new0,  SNode *new1) {
      if (new0 != Null) add_to_rc(new0, 1);
      if (new1 != Null) add_to_rc(new1, 1);
      if (DCAS(A0, A1, old0, old1, new0, new1)) {
            LFRCDestroy(old0, old1);
            return true;
      } else {
            LFRCDestroy(new0, new1);
            return false;
      }
}
```

where the LFRCDCAS operation accepts six parameters, corresponding to the DCAS parameters described earlier. The illustrated implementation of the LFRCDCAS operation is similar to that of the LFRCStore operation in that it increments the reference counts of objects before creating new pointers to them (lines 4-5) using the add_to_rc function, thereby temporarily setting these counts artificially high. However, the LFRCDCAS operation differs from the LFRCStore operation in that it does not insist on eventually creating those new pointers. If the DCAS at line 6 fails, then LFRCDCAS calls LFRCDestroy for each of the objects whose reference counts were previously incremented, thereby compensating for the previous increments and then returning false (*see* lines 9-11). On the other hand, if the DCAS succeeds, then the previous increments were justified but we have destroyed two pointers, namely the previous values of the two locations updated by the DCAS. Therefore, the LFRCDCAS operation calls LFRCDestroy to decrement the reference counts of (and potentially free) the corresponding objects and then returns true (*see* lines 7-8). One suitable implementation of an LFRCCAS operation (not shown) is just a simplification of the LFRCDCAS with handling of the second location omitted.

While the above-described LFRCLoad, LFRCStore, LFRCDCAS and LFRCDestroy operations are sufficient for the illustrated implementation (or transformation) of pushRightpopRight access operation (and their left-end analogues) for a concurrent shared object implementation of a deque, other access operations, other implementations and other concurrent shared objects may employ alternative or additional LFRC pointer operations. For example, in some implementations, an LFRCCopy operation may be useful. One implementation is as follows:

```
void LFRCCopy(SNode **v, SNode *w) {
      if (w != Null)
            add_to_rc(w,1);
      LFRCDestroy(*v);
      *v = w;
}
```

where the LFRCCopy operation accepts two parameters, a pointer v to a local pointer variable, and a value w of a local pointer variable. The semantics of this operation is to assign the value w to the variable pointed to by v. This creates a new pointer to the object referenced by w (if w is not NULL), so LFRCCopy increments the reference count of that object (lines 2-3). The LFRCCopy operation also destroys a pointer, namely the previous contents of *v, so LFRCCopy calls LFRCDestroy (line 4) to decrement the reference count of the object referenced by the now-destroyed pointer. Finally, LFRCCopy assigns the value w to the pointer variable pointed to by v and returns.

Other LFRC operations that may be useful in some implementations include a variant of the previously described LFRCLoad operation suitable for use in situations where the target of the load cannot contain a pointer. For example, such a variation may be implemented as follows:

```
void LFRCLoad(SNode **A) {
  SNode *a;
  long r;
  while (true) {
    a = *A;
    if (a == NULL)
      return NULL;
    r = a->rc;
    if (DCAS(A,&a->rc,a,r,a,r+1))
      return a;
  }
}
```

Other suitable LFRC operations include the previously-described LFRCStoreAlloc operation, which may be implemented as follows:

```
void LFRCStoreAlloc(SNode **A, SNode *v) {
  SNode *oldval;
  while (true) {
    oldval = *A;
    if (CAS(A,oldval,v)) {
      LFRCDestroy (oldval);
      return;
    }
  }
}
```

in situations in which we want to invoke an allocation routine directly as the second parameter of a LFRC store operation. In addition, some implementations or transformations may include a variant of the LFRCDCAS operation such as the following:

```
bool LFRCDCAS1(SNode **a0, int *a1, SNode *old0, int old1,
SNode *new0, int new1){

  if (new0 != NULL)
    add_to_rc(new0,1);

  if (DCAS(a0,a1,old0,old1,new0,new1)) {    // Do DCAS
    LFRCDestroy(old0);
    return true;
  } else {
    LFRCDestroy(new0);
    return false;
  }
}
```

where the second location operated upon by the DCAS pointer operation contains a literal (e.g., an integer) rather than a pointer.

Some implementations or transformations may exploit other LFRC pointer operations such as the previously described `LFRCPass` operation, which may be implemented as follows:

```
SNode* LFRCPass(SNode *p) {
  if (p!=NULL)
    add_to_rc(p,1);
  return p;
}
```

where the `LFRCPass` function may be employed to facilitate passing a pointer by value while appropriately maintaining a corresponding reference count. These and other variations on the illustrated set of LFRC pointer operations will be appreciated by persons of ordinary skill in the art based on the description herein.

While the invention has been described with reference to various embodiments, it will be understood that these embodiments are illustrative and that the scope of the invention is not limited to them. Terms such as always, never, all, none, etc. are used herein to describe sets of consistent states presented by a given computational system. Of course, persons of ordinary skill in the art will recognize that certain transitory states may and do exist in physical implementations even if not presented by the computational system. Accordingly, such terms and invariants will be understood in the context of consistent states presented by a given computational system rather than as a requirement for precisely simultaneous effect of multiple state changes. This "hiding" of internal states is commonly referred to by calling the composite operation "atomic", and by allusion to a prohibition against any process seeing any of the internal states partially performed.

Many variations, modifications, additions, and improvements are possible. For example, while various full-function deque realizations have been described in detail, realizations implementations of other shared object data structures, including realizations that forgo some of access operations, e.g., for use as a

FIFO, queue, LIFO, stack or hybrid structure, will also be appreciated by persons of ordinary skill in the art. In addition, more complex shared object structures may be defined that exploit the techniques described herein. Other synchronization primitives may be employed and a variety of distinguishing pointer values may be employed including without limitation, the self-referencing pointer values, marker node pointers and null pointers employed in some realizations described herein. In general, the particular data structures, synchronization primitives and distinguishing pointer values employed are implementation specific and, based on the description herein, persons of ordinary skill in the art will appreciate suitable selections for a given implementation.

Plural instances may be provided for components, operations or structures described herein as a single instance. Finally, boundaries between various components, operations and data stores are somewhat arbitrary, and particular operations are illustrated in the context of specific illustrative configurations. Other allocations of functionality are envisioned and may fall within the scope of claims that follow. Structures and functionality presented as discrete components in the exemplary configurations may be implemented as a combined structure or component. These and other variations, modifications, additions, and improvements may fall within the scope of the invention as defined in the claims that follow.

WHAT IS CLAIMED IS:

1. A method of facilitating non-blocking access to a double-ended queue (deque) encoded using a doubly-linked-list of nodes and opposing-end identifiers, the method comprising:

defining linearizable push and pop operations operable on opposing-ends of the deque, wherein, for at least those deque states of two or more nodes, opposing-end executions of the pop operation include a linearizable synchronization operation on disjoint pairs of storage locations;

for at least those deque states of exactly two nodes, handling potentially concurrent execution of opposing-end pop operations by encoding a distinguishing pointer value in a popped node; and

treating presence of the distinguishing pointer value in a node identified by one of the end identifiers as indicative of a logically empty deque.

2. The method of claim 1,

wherein the distinguishing pointer value includes a self-referencing pointer value.

3. The method of claim 1,

wherein the distinguishing pointer value includes a null value.

4. The method of claim 1,

wherein the distinguishing pointer value identifies a marker node.

5. The method of claim 1,

wherein the linearizable synchronization operation is a double compare and swap operation.

6. The method of claim 1,

wherein the linearizable synchronization operation employs transactional memory.

7. The method of claim 1, wherein for those deque states of two or more nodes,

the linearizable synchronization operation of left-end executions of the pop operation operates on a left-end one of the end identifiers and a right pointer of the node instantaneously identified thereby; and

the linearizable synchronization operation of right-end executions of the pop operation operates on a right-end one of the end identifiers and a left pointer of the node instantaneously identified thereby.

8. The method of claim 1,

wherein an execution environment includes automatic reclamation of storage; and

wherein execution of the pop operations includes severing a pointer chain to a previously popped node so that the severed, previously popped node may be reclaimed by the automatic reclamation of storage.

9. The method of claim 1, wherein execution of one of the pop operations includes:
severing a pointer chain to the node popped thereby; and
explicitly reclaiming the severed node.

10. The method of claim 9,
wherein the push and pop operations employ lock free reference counting pointer operations.

11. The method of claim 9, including the explicit reclamation, employed in the implementation of garbage collector.

12. A double-ended queue (deque) representation comprising:
a doubly-linked list encoded in addressable storage;
left- and right-end identifiers for respective ends to the list; and
a computer readable encoding of opposing end push and pop operations,
wherein executions of the opposing end pop operations are disjoint with respect to each other for deque states of two or more nodes, and wherein handling of potentially concurrent execution of the opposing end pop operations for a deque state of exactly two nodes includes encoding a distinguishing pointer value in a popped node and treating presence of the distinguishing pointer value in a node identified by one of the left- and right-end identifiers as indicative of a logically empty deque.

13. The deque representation of claim 12, wherein the opposing-end pop operations employ implementations of a linearizable synchronization primitive to mediate:
concurrent execution of same-end instances of the push and pop operations; and
for single-node deque states, concurrent execution of opposing end instances of the push and pop operations.

14. The deque representation of claim 12,
wherein the distinguishing pointer value includes one of a self-pointer and a null value.

15. The deque representation of claim 12,
wherein the distinguishing pointer value identifies a marker node.

16. The deque representation of claim 12,
wherein the addressable storage is managed by a garbage collector; and

wherein execution of pop operations includes severing a pointer chain to a previously popped node so that the severed, previously popped node may be reclaimed by the garbage collector.

17. The deque representation of claim 12,
wherein the push and pop operations employ lock-free reference counting pointer operations; and
wherein execution of the push and pop operations includes explicit reclamation of a previously popped node severed from the list.

18. The method of managing access to a doubly-linked list of nodes susceptible to concurrent removals from left and right ends thereof, the method comprising:
executing as part of a left remove-type operation, a linearizable synchronization operation to store a distinguishing pointer value in a right list pointer of a node removed thereby and to update a left-end identifier to identify a node to the right of the removed node; and
executing as part of a right remove-type operation, a linearizable synchronization operation to store a distinguishing pointer value in a left list pointer of a node removed thereby and to update a right-end identifier to identify a node to the left of the removed node,
wherein concurrent execution of left and right remove-type operations on a two-node state of the list is tolerated, at least in part, through treatment of the distinguishing pointer value in a right list pointer of a node identified by the right-end identifier or in a left list pointer of a node identified by the left-end identifier as indicative of a logically empty state of the list.

19. The method of claim 18,
wherein the doubly-linked list of nodes encodes a double ended queue (deque);
wherein the left and right remove-type operations are left and right pop operations, respectively; and
further comprising:
executing as part of a left push operation, a linearizable synchronization operation to store a pointer to a new node into both the left-end identifier and a left list pointer of the list node previously identified thereby; and
executing as part of a right push operation, a linearizable synchronization operation to store a pointer to a new node into both the right-end identifier and a right list pointer of the list node previously identified thereby.

20. The method of claim 18, further comprising:
executing as part of a insert-type operation, a linearizable synchronization operation to store a pointer to a new node into both a respective one of the left- and right-end identifiers and a respective list pointer of the list node previously identified thereby.

21. A concurrent shared object representation comprising:
a sequence of zero or more values encoded in computer readable form as a doubly-linked list of nodes, each having a pair of opposing-direction list pointers; and

linearizable, non-blocking access operations defined for access to each of opposing ends of the sequence, the linearizable, non-blocking access operations including at least remove-type operations,

wherein concurrent execution of opposing-end ones of the remove-type operations on a two-node state of the sequence is tolerated by employing a linearizable synchronization primitive to store a distinguishing pointer value in a respective one of the list pointers of a removed node and to update a respective one of opposing-end identifiers to identify a node adjacent to the removed node, and

wherein presence of the distinguishing pointer value in a node identified by one of the opposing-end identifiers encodes a logically empty state of the concurrent shared object.

22. The concurrent shared object representation of claim 21,

wherein the linearizable, non-blocking access operations further include an insert-type operation defined to operate at one of the ends of the sequence.

23. The concurrent shared object representation of claim 21,

wherein the linearizable, non-blocking access operations further include insert-type operations defined to operate at each of the opposing ends of the sequence.

24. The concurrent shared object representation of claim 21, wherein the linearizable synchronization primitive includes one of:

a multi-way compare and swap; and

use of a transactional memory facility.

25. A computer program product encoded in at least one computer readable medium, the computer program product comprising:

functional sequences implementing left- and right-end access operations on a double-ended concurrent shared object, the concurrent shared object instantiable as a doubly-linked-list delimited by a pair of left- and right-end identifiers,

wherein instances of the functional sequences concurrently executable by plural execution units and each including a linearizable synchronization operation to mediate competing executions of the functional sequences, and

wherein, for at least two-node states of the concurrent shared object, concurrent execution of left- and right-end remove-type access operations is handled by encoding a distinguishing pointer value in removed nodes and treating presence of the distinguishing pointer value in a node identified by a respective one of the end identifiers as indicative of a logically empty state of the concurrent shared object.

26. The computer program product of claim 25, wherein the access operations include:

the left- and right-end remove-type operations; and

at least one insert-type operation.

27. The computer program product of claim 25, wherein the access operations include left- and right-end push and pop operations.

28. The computer program product of claim 25, further comprising:
functional sequences implementing a garbage collector and employing instantiations of the double-ended concurrent shared object in coordination thereof.

29. The computer program product of 25,
wherein the at least one computer readable medium is selected from the set of a disk, tape or other magnetic, optical, or electronic storage medium and a network, wireline, wireless or other communications medium.

30. An apparatus comprising:
plural processors;
a one or more stores addressable by the plural processors;
first- and second-end identifiers accessible to each of the plural processors for identifying opposing ends of a doubly-linked list of nodes encoded in the one or more stores; and
means for coordinating competing opposing-end pop operations on the doubly-linked list, including a two-node state thereof, the coordinating means employing instances of a linearizable synchronization operation and a distinguishing pointer value encoding indicative of a logically empty state of the doubly-linked list.

31. The apparatus of claim 30,
means for explicitly reclaiming a node severed from the list.

32. A double-ended concurrent shared object organized as a bi-directional referencing chain and including dynamic allocation of nodes thereof, the double-ended concurrent shared object employing a distinguishing pointer value to indicate an empty state thereof and supporting concurrent non-interfering opposing-end accesses for states of two or more nodes.

33. The double-ended concurrent shared object of claim 32, embodied as a doubly-linked list of nodes allocated from a shared memory of a multiprocessor and access operations executable by processors thereof.

34. The double-ended concurrent shared object of claim 32, embodied as a computer program product encoded in media, the computer program product defining a data structure instantiable in shared memory of a multiprocessor and instructions executable thereby implementing access operations.

35. The double-ended concurrent shared object of claim 34,
wherein the data structure includes a double-ended queue; and
wherein the access operations include opposing-end variants of push and pop operations.

36. The double-ended concurrent shared object of claim 32,
wherein those of the nodes that are severed from the referencing chain are explicitly reclaimed by operation of respective ones of the concurrent non-interfering opposing-end accesses.

37. The double-ended concurrent shared object of claim 32,
wherein those of the nodes that are severed from the referencing chain are reclaimed by an automatic storage reclamation facility of an execution environment.

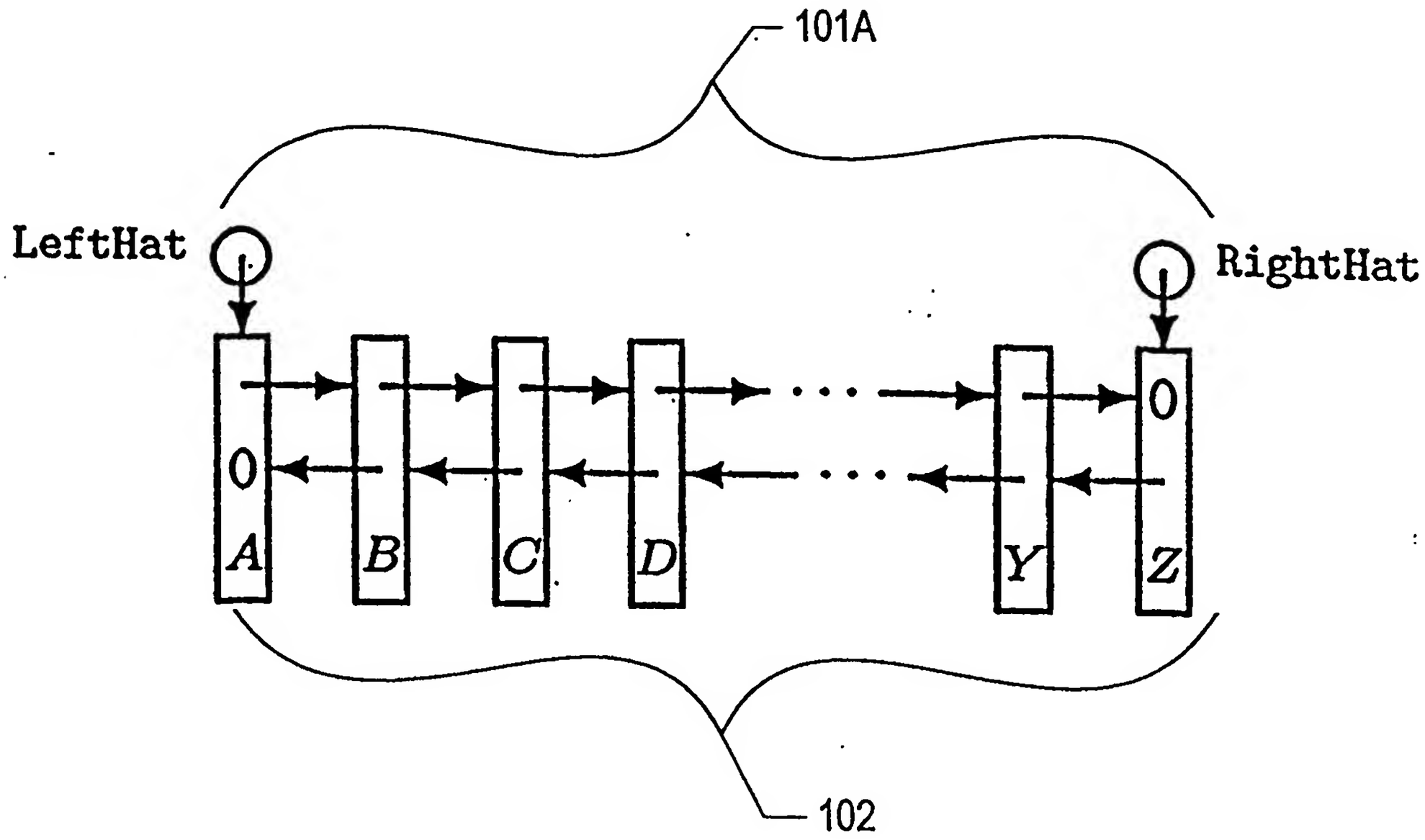

101A
LeftHat
RightHat
0
A
B
C
D
Y
0
Z
102

FIG. 1A

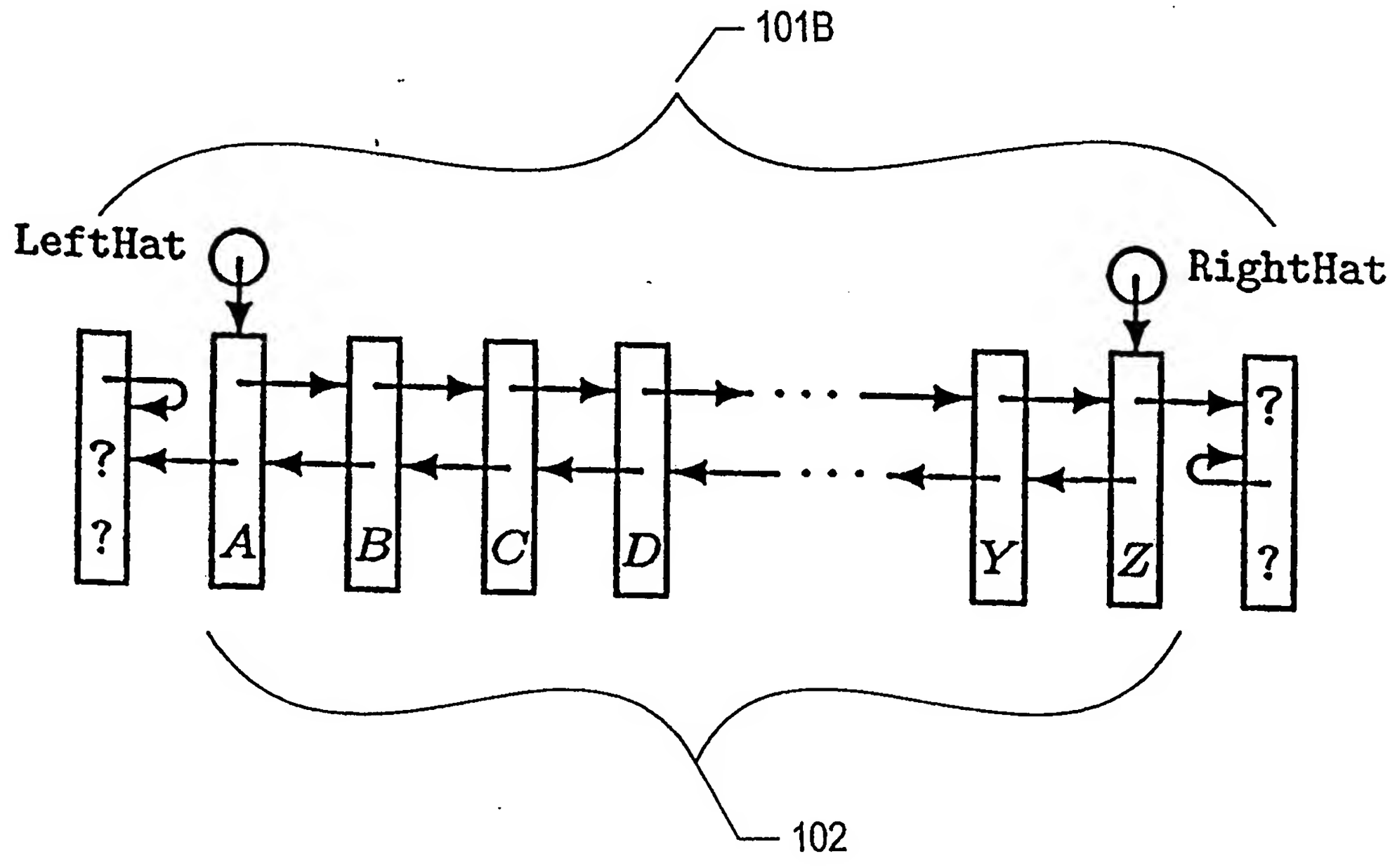

101B
LeftHat
RightHat
?
?
A
B
C
D
Y
Z
?
?
102

FIG. 1B

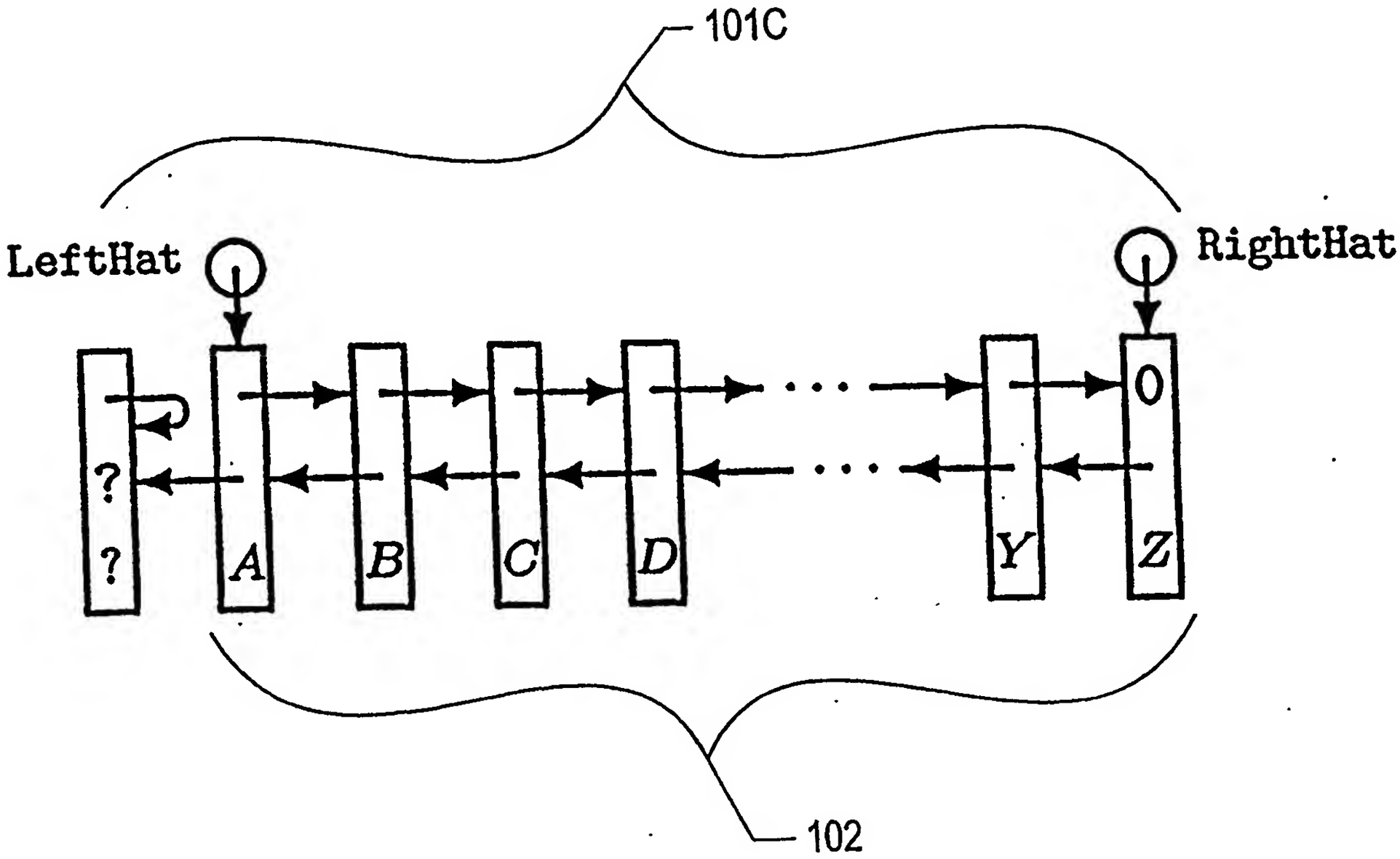

101C
LeftHat
RightHat
?
?
A
B
C
D
Y
Z
0
102

FIG. 1C

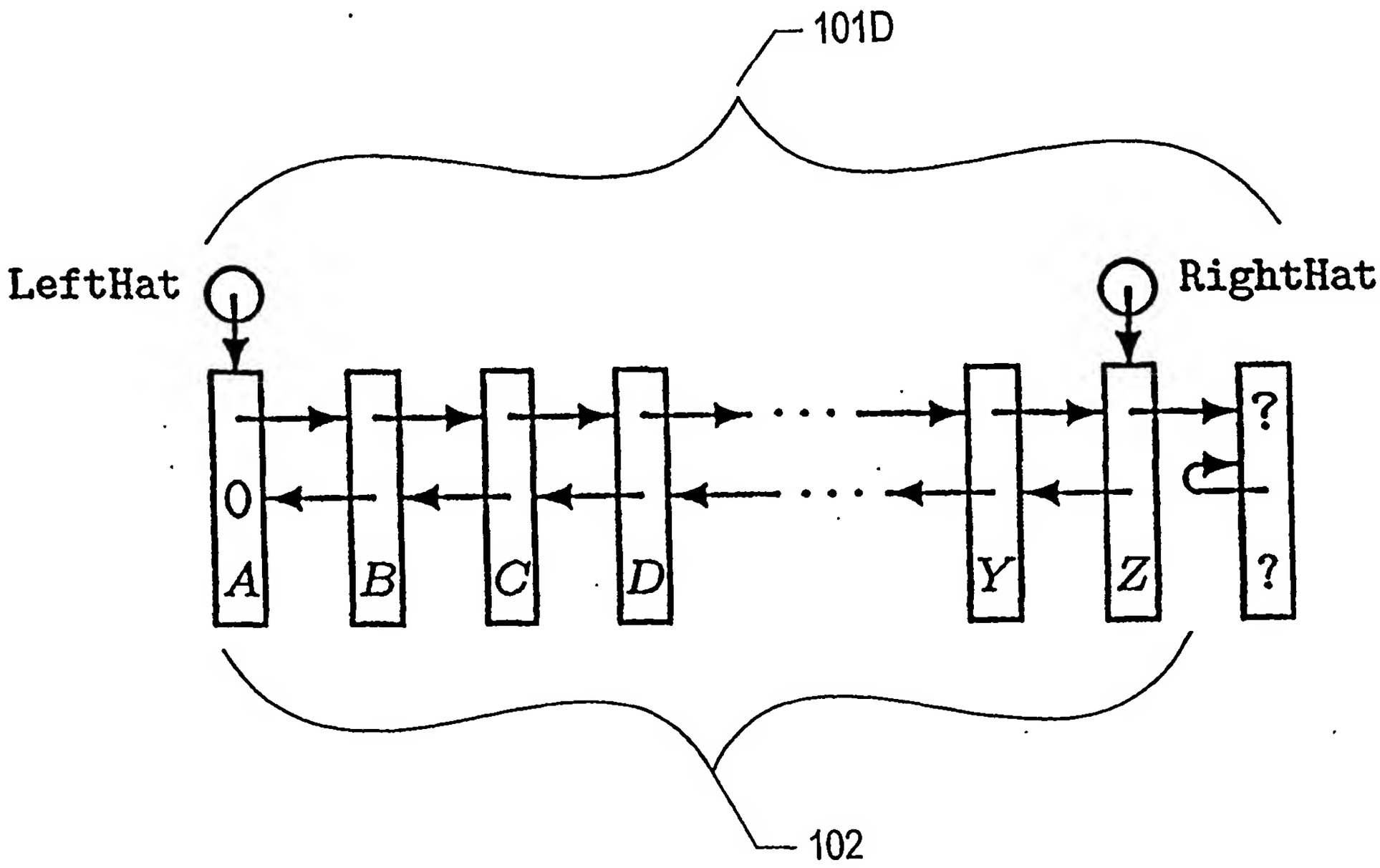

101D
LeftHat
RightHat
0
A
B
C
D
Y
Z
?
?
102

FIG. 1D

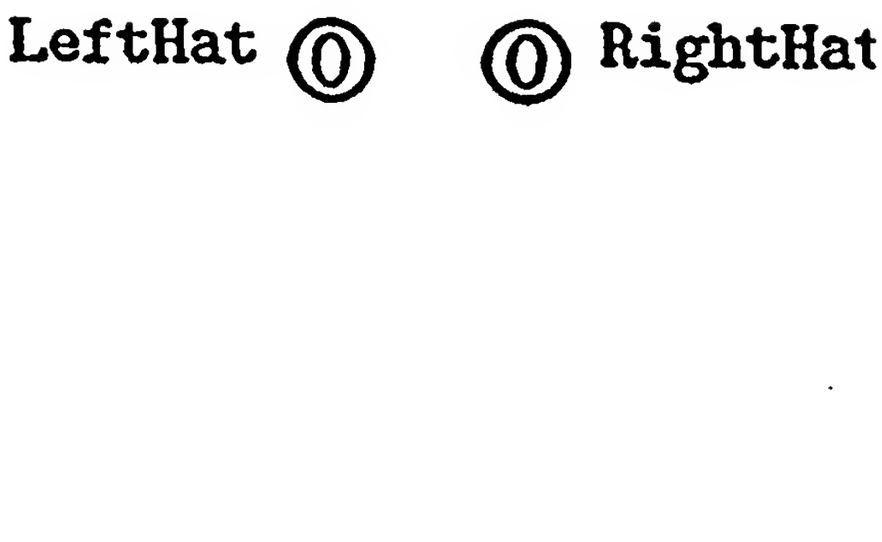


FIG. 2A

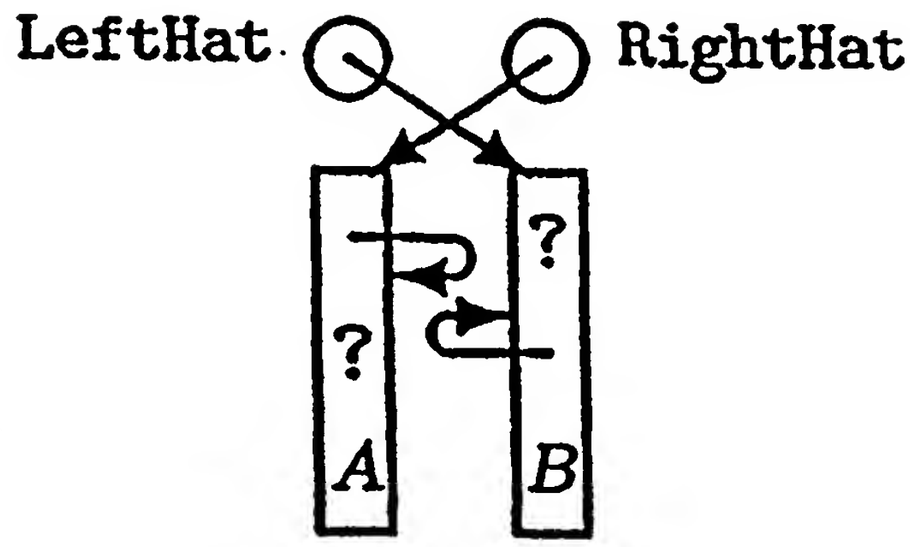


FIG. 2B

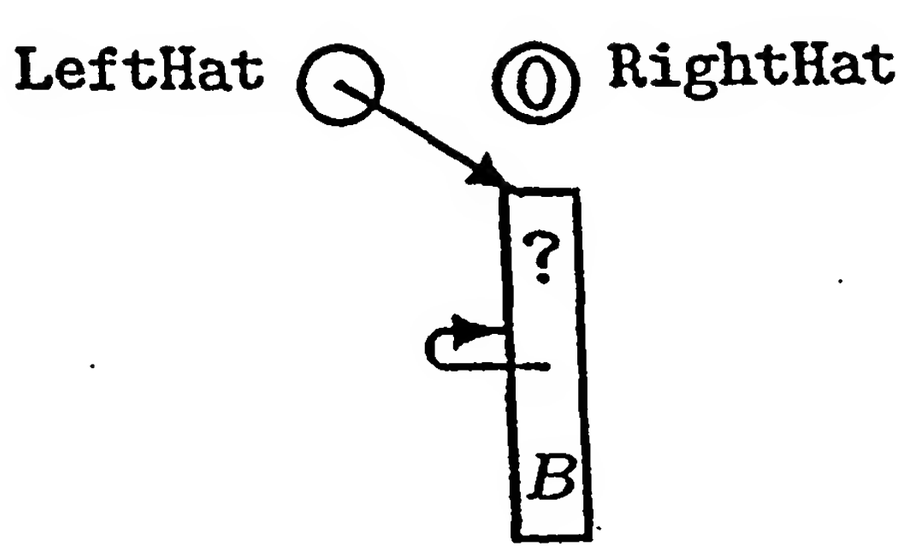


FIG. 2C

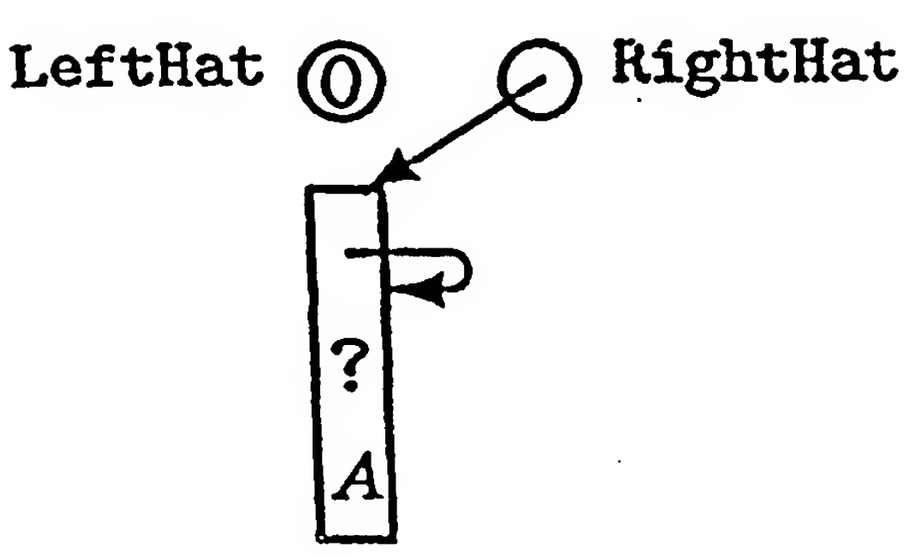


FIG. 2D

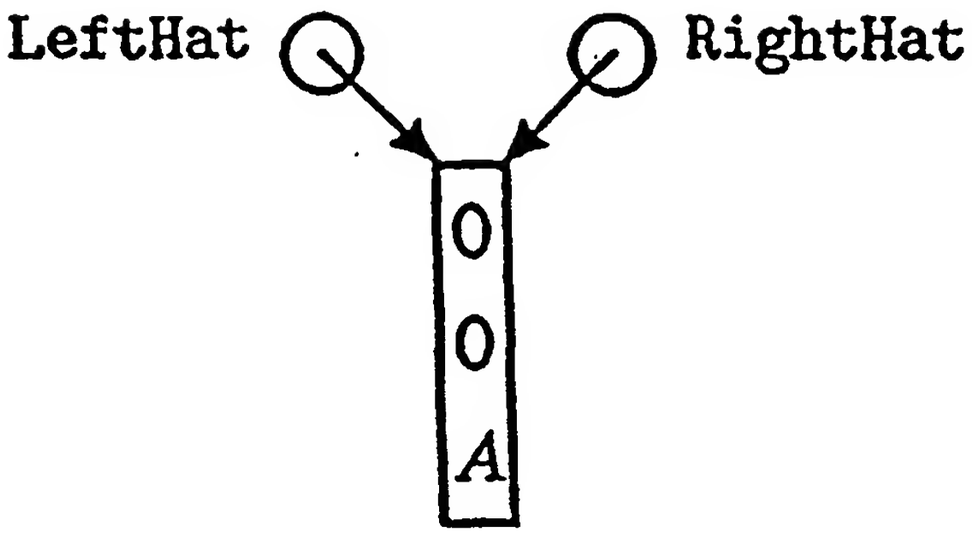


FIG. 3A

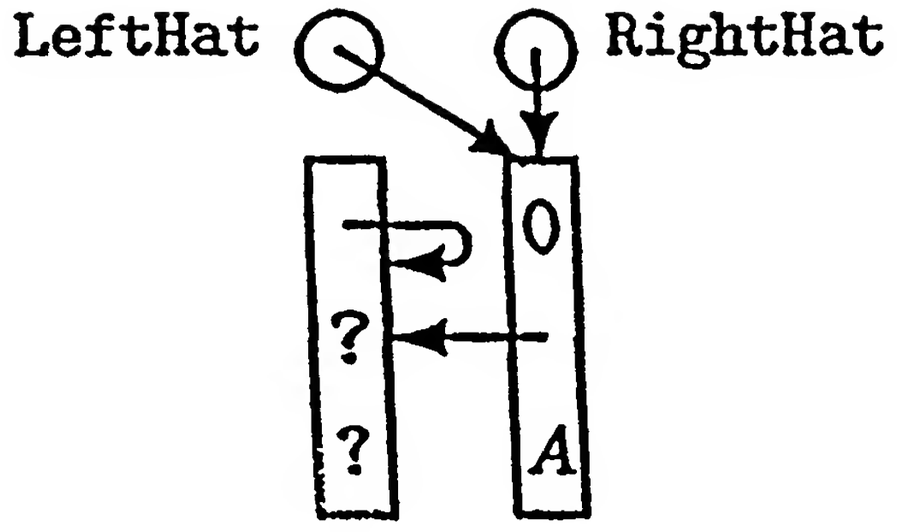


FIG. 3B

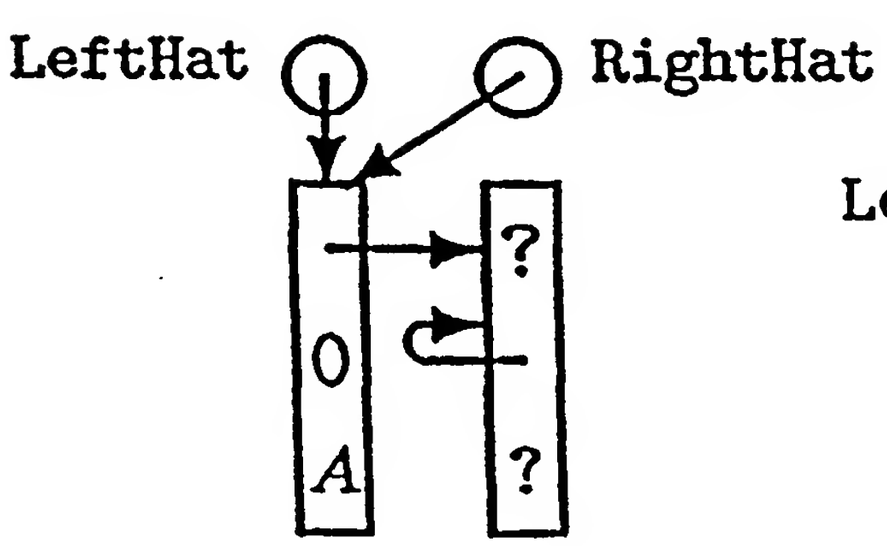


FIG. 3C

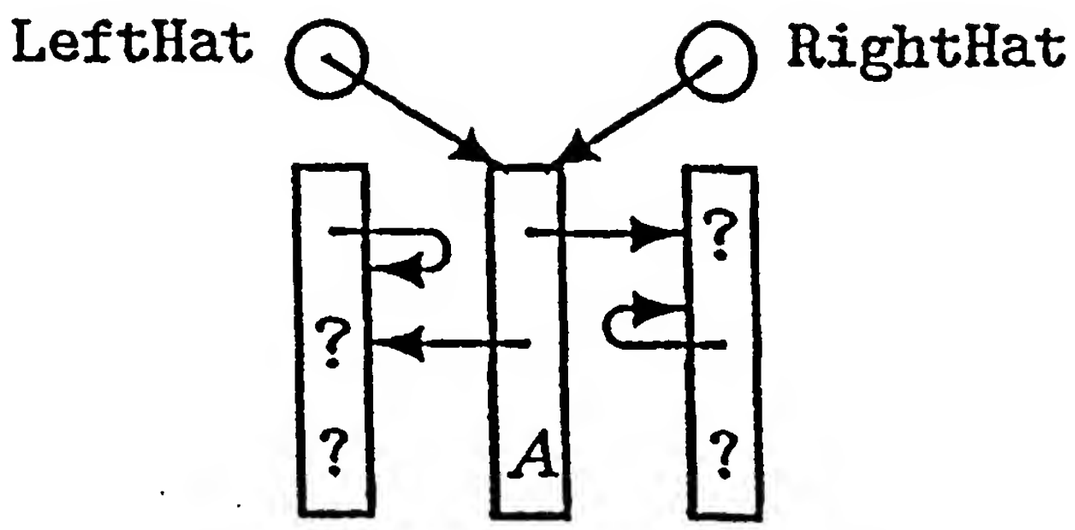


FIG. 3D

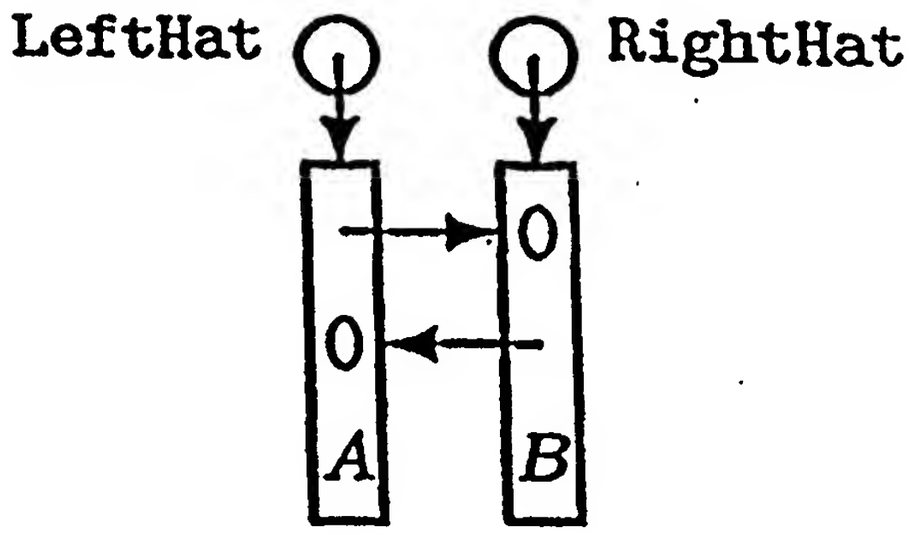


FIG. 4A

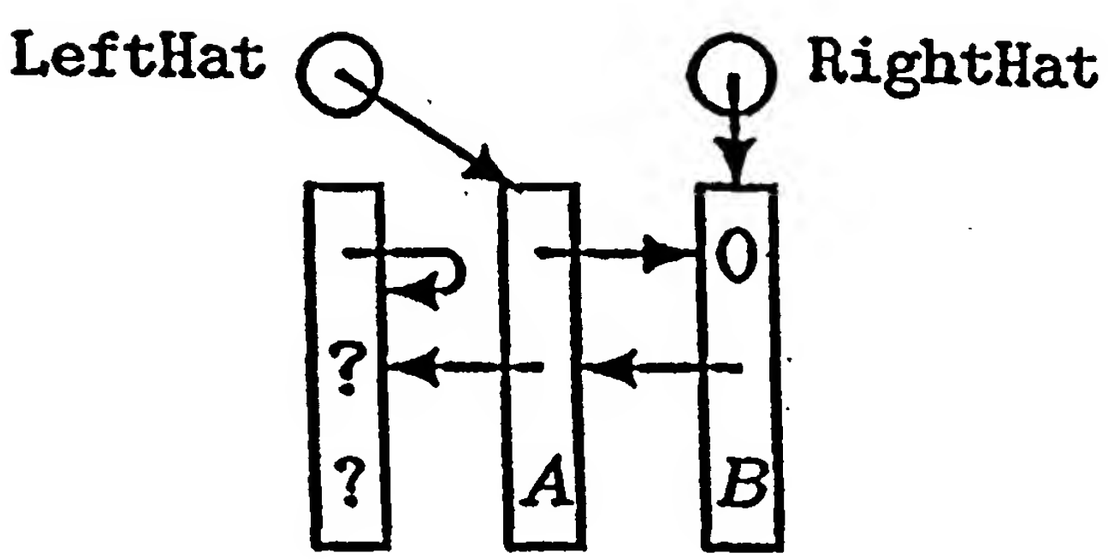


FIG. 4B

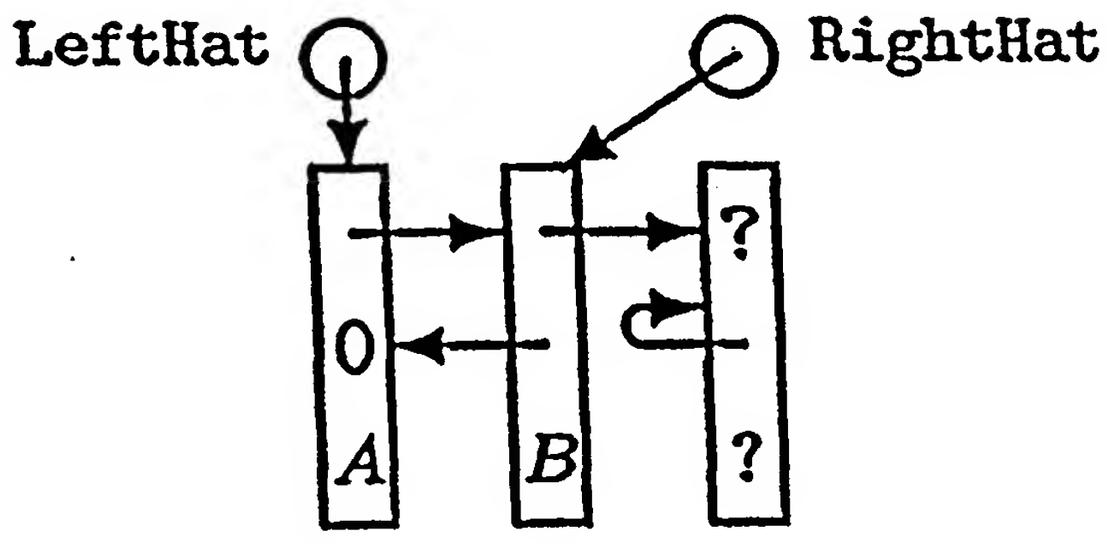


FIG. 4C

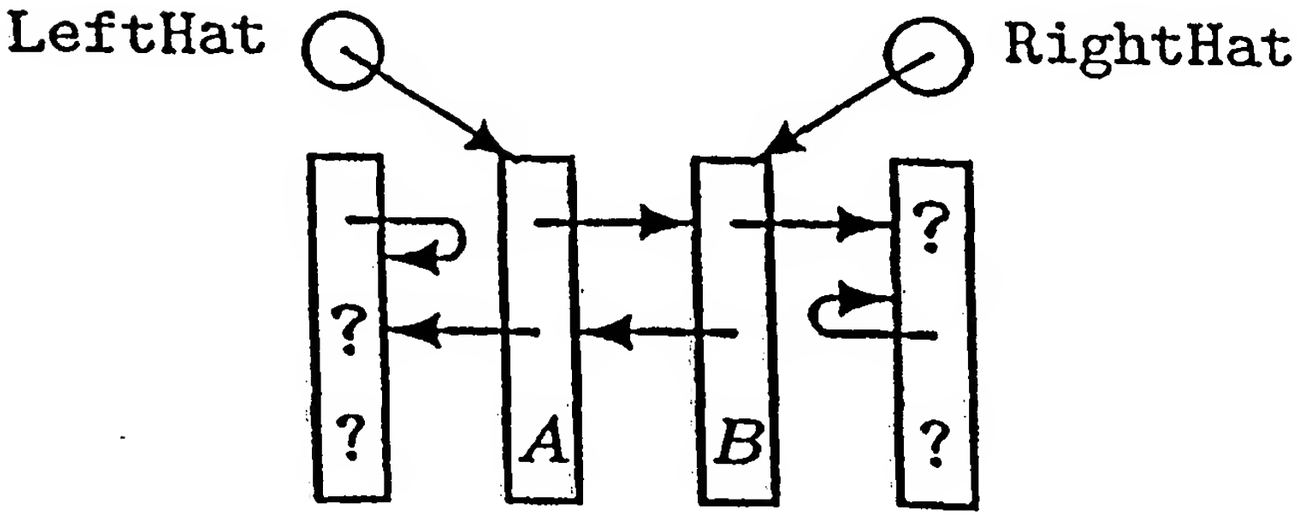


FIG. 4D

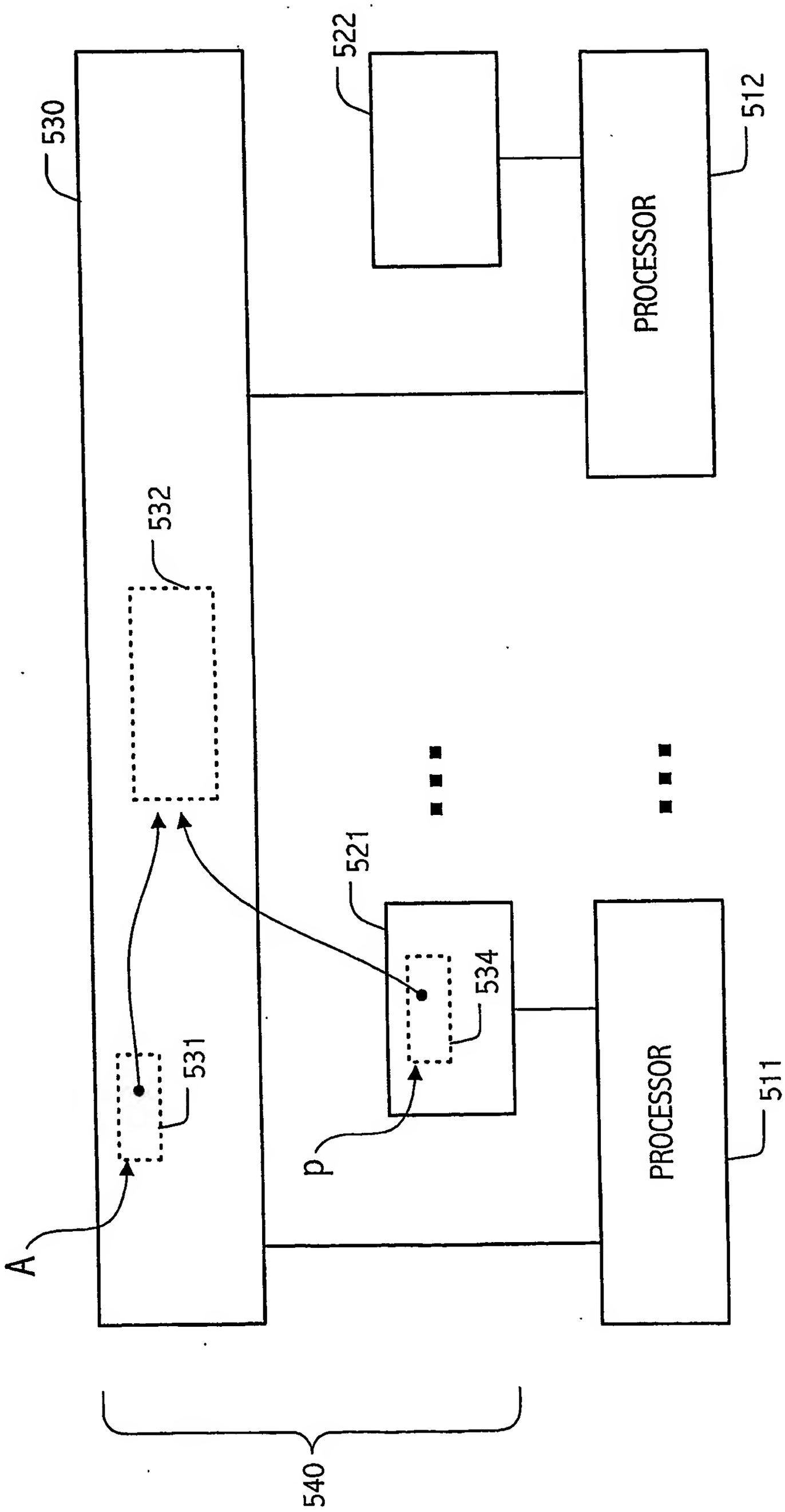


FIG. 5